Tivoli® Directory Integrator
Version 7.1

Reference Guide

SC27-2707-00

Tivoli® Directory Integrator
Version 7.1

Reference Guide

SC27-2707-00

Note

Note: Before using this information and the product it supports, read the general information under Appendix F, "Notices," on page 613.

Product Version 7.1

This edition applies to version 7.1 of the IBM Tivoli Directory Integrator and to all subsequent releases and modifications until otherwise indicated in new editions.

Preface

This document contains the information that you need to develop solutions using components that are part of the IBM® Tivoli® Directory Integrator.

Who should read this book

This book is intended for those responsible for the development, installation and administration of solutions with the IBM Tivoli Directory Integrator.

IBM Tivoli Directory Integrator components are designed for network administrators who are responsible for maintaining user directories and other resources. This document assumes that you have practical experience installing and using both IBM Tivoli Directory Integrator, and the reader should be familiar with the concepts and the administration of the systems that the developed solution will connect to. Depending on the solution, these could include, but are not limited to, one or more of the following products, systems and concepts:

- IBM Directory Server
- IBM Tivoli Identity Manager
- IBM Java™ Runtime Environment (JRE) or Sun Java Runtime Environment
- Microsoft® Active Directory
- PC and UNIX® operating systems
- Security management
- Internet protocols, including HTTP, HTTPS and TCP/IP
- Lightweight Directory Access Protocol (LDAP) and directory services
- A supported user registry
- Authentication and authorization
- SAP ABAP Application Server.

Publications

Read the descriptions of the IBM Tivoli Directory Integrator library and the related publications to determine which publications you might find helpful. After you determine the publications you need, refer to the instructions for accessing publications online.

IBM Tivoli Directory Integrator library

The publications in the Tivoli Directory Integrator library are:

IBM Tivoli Directory Integrator V7.1 Getting Started
: A brief tutorial and introduction to Tivoli Directory Integrator 7.1. Includes examples to create interaction and hands-on learning of IBM Tivoli Directory Integrator.

IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide
: Includes complete information about installing, migrating from a previous version, configuring the logging functionality, and the security model underlying the Remote Server API of IBM Tivoli Directory Integrator. Contains information on how to deploy and manage solutions.

IBM Tivoli Directory Integrator V7.1 Users Guide
: Contains information about using IBM Tivoli Directory Integrator 7.1. Contains instructions for designing solutions using the Tivoli Directory Integrator designer tool (**ibmditk**) or running the ready-made solutions from the command line (**ibmdisrv**). Also provides information about interfaces, concepts and AssemblyLine creation.

IBM Tivoli Directory Integrator V7.1 Reference Guide
Contains detailed information about the individual components of IBM Tivoli Directory Integrator 7.1: Connectors, Function Components, Parsers and so forth – the building blocks of the AssemblyLine.

IBM Tivoli Directory Integrator V7.1 Problem Determination Guide
Provides information about IBM Tivoli Directory Integrator 7.1 tools, resources, and techniques that can aid in the identification and resolution of problems.

IBM Tivoli Directory Integrator V7.1 Messages Guide
Provides a list of all informational, warning and error messages associated with the IBM Tivoli Directory Integrator 7.1.

IBM Tivoli Directory Integrator V7.1 Password Synchronization Plug-ins Guide
Includes complete information for installing and configuring each of the five IBM Password Synchronization Plug-ins: Windows Password Synchronizer, Sun Directory Server Password Synchronizer, IBM Directory Server Password Synchronizer, Domino Password Synchronizer and Password Synchronizer for UNIX and Linux®. Also provides configuration instructions for the LDAP Password Store and JMS Password Store.

IBM Tivoli Directory Integrator V7.1 Release Notes
Describes new features and late-breaking information about IBM Tivoli Directory Integrator 7.1 that did not get included in the documentation.

Related publications

Information related to the IBM Tivoli Directory Integrator is available in the following publications:

- IBM Tivoli Directory Integrator 7.1 uses the JNDI client from Sun Microsystems. For information about the JNDI client, refer to the *Java Naming and Directory Interface™ Specification* on the Sun Microsystems Web site at http://java.sun.com/j2se/1.5.0/docs/guide/jndi/index.html.
- The Tivoli Software Library provides a variety of Tivoli publications such as white papers, datasheets, demonstrations, redbooks, and announcement letters. The Tivoli Software Library is available on the Web at: http://www.ibm.com/software/tivoli/library/
- The *Tivoli Software Glossary* includes definitions for many of the technical terms related to Tivoli software. The *Tivoli Software Glossary* is available on the World-Wide Web, in English only, athttp://publib.boulder.ibm.com/tividd/glossary/tivoliglossarymst.htm

Accessing publications online

The publications for this product are available online in Portable Document Format (PDF) or Hypertext Markup Language (HTML) format, or both in the Tivoli software library: http://www.ibm.com/software/tivoli/library.

To locate product publications in the library, click the **Product manuals** link on the left side of the Library page. Then, locate and click the name of the product on the Tivoli software information center page.

Information is organized by product and includes READMEs, installation guides, user's guides, administrator's guides, and developer's references as necessary.

Note: To ensure proper printing of PDF publications, select **Fit to page** in the Adobe Acrobat Print window (which is available when you click **File->Print**).

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully. With Tivoli Directory Integrator 7.1, you can use assistive technologies to hear and navigate the interface. After installation you also can use the keyboard instead of the mouse to operate all features of the graphical user interface.

Accessibility features

The following list includes the major accessibility features in Tivoli Directory Integrator 7.1:

- Supports keyboard-only operation.
- Supports interfaces commonly used by screen readers.
- Discerns keys as tactually separate, and does not activate keys just by touching them.
- Avoids the use of color as the only way to communicate status and information.
- Provides accessible documentation.

Keyboard navigation

This product uses standard Microsoft Windows® navigation keys for common Windows actions such as access to the File menu, copy, paste, and delete. Actions that are unique to Tivoli Directory Integrator use Tivoli Directory Integrator keyboard shortcuts. Keyboard shortcuts have been provided wherever needed for all actions.

Interface Information

The accessibility features of the user interface and documentation include:

- Steps for changing fonts, colors, and contrast settings in the Configuration Editor:
  1. Type `Alt-W` to access the Configuration Editor **Window** menu. Using the downward arrow, select **Preferences...** and press `Enter`.
  2. Under the **Appearance** tab, select **Colors and Fonts** settings to change the fonts for any of the functional areas in the Configuration Editor.
  3. Under **View and Editor Folders**, select the colors for the Configuration Editor, and by selecting colors, you can also change the contrast.
- Steps for customizing keyboard shortcuts, specific to IBM Tivoli Directory Integrator:
  1. Type `Alt-W` to access the Configuration Editor **Window** menu. Using the downward arrow, select **Preferences...** .
  2. Using the downward arrow, select the General category; right arrow to open this, and type downward arrow until you reach the entry **Keys**.

     Underneath the **Scheme** selector, there is a field, the contents of which say "type filter text." Type `tivoli directory integrator` in the filter text field. All specific Tivoli Directory Integrator shortcuts are now shown.
  3. Assign a keybinding to any Tivoli Directory Integrator command of your choosing.
  4. Click **Apply** to make the change permanent.

The Configuration Editor is a specialized instance of an Eclipse workbench. More detailed information about accessibility features of applications built using Eclipse can be found at http://help.eclipse.org/help33/topic/org.eclipse.platform.doc.user/concepts/accessibility/accessmain.htm

- The information center and its related publications are accessibility-enabled for the JAWS screen reader and the IBM Home Page Reader. You can operate all documentation features using the keyboard instead of the mouse.

Vendor software

The IBM Tivoli Directory Integrator installer uses the InstallAnywhere 2009 (IA) installer technology.

The IBM Tivoli Directory Integrator 7.1 installer has accessibility features that are independent from the product. The installer supports 3 UI modes:

GUI Keyboard-only operation is supported in GUI mode, and the use of a screen reader is possible. In order to get the most from a screen reader, you should use the Java Access Bridge and launch the installer with a Java access Bridge enabled JVM, for example:

```
install_tdiv71_win_x86.exe LAX_VM "Java_DIR/jre/bin/java.exe"
```

The JVM used should be a Java 6 JRE.

Console
In console mode, keyboard-only operation is supported and all displays and user options are displayed as text that can be easily read by screen readers. Console mode is the suggested install method for accessibility.

Silent In silent mode, user responses are given through a response file, and no user interaction is required.

Related accessibility information

Visit the *IBM Accessibility Center* at http://www.ibm.com/able for more information about IBM's commitment to accessibility.

Contacting IBM Software support

Before contacting IBM Tivoli Software support with a problem, refer to IBM System Management and Tivoli software Web site at:

http://www.ibm.com/software/sysmgmt/products/support/IBMDirectoryIntegrator.html

If you need additional help, contact software support by using the methods described in the *IBM Software Support Handbook* at the following Web site:

http://techsupport.services.ibm.com/guides/handbook.html

The guide provides the following information:

- Registration and eligibility requirements for receiving support
- Telephone numbers and e-mail addresses, depending on the country in which you are located
- A list of information you must gather before contacting customer support

A list of most requested documents as well as those identified as valuable in helping answer your questions related to IBM Tivoli Directory Integrator can be found at http://www-01.ibm.com/support/docview.wss?rs=697&uid=swg27009673.

Contents

Preface iii
Who should read this book iii
Publications iii
IBM Tivoli Directory Integrator library iii
Related publications iv
Accessing publications online iv
Accessibility iv
Accessibility features v
Keyboard navigation v
Interface Information v
Vendor software v
Related accessibility information vi
Contacting IBM Software support vi

Chapter 1. Introduction 1

Chapter 2. Connectors 3
Connector availability and reference 3
Connector Interfaces 3
Script-based Connectors 6
Configurations. 6
Connector re-use 6
Active Directory Change Detection Connector . . . 8
Tracking changes in Active Directory 8
Change detection 9
Using the Active Directory Change Detection Connector 10
Configuration 12
See also. 13
AssemblyLine Connector 14
Configuration 14
Using the Connector 15
See also. 17
Axis Easy Web Service Server Connector 19
Hosting a WSDL file 20
Schema 20
Configuration 21
Connector Operation 23
See also. 23
Axis2 Web Service Server Connector 25
Comparison between Axis1 and Axis2 components 25
Using the Connector 26
Schema 28
Configuration 33
Security and Authentication 34
See also. 35
Command line Connector 37
Native-encoded output on some operating systems. 37
Some words on quoting 37
Configuration 38
Examples 38
See also. 38
Database Connector 39
Configuration 39
Direct TCP /URL scripting 41
TCP 41
URL 41
Domino/Lotus Notes Connectors 43
Session types 43
Post Install Configuration. 45
Native API call threading. 46
The ncso.jar file 47
"Classes" folder 48
Domino Change Detection Connector. 49
Domino Users Connector 59
Domino AdminP Connector 73
Lotus Notes Connector 77
See also. 81
ITIM DSMLv2 Connector 83
Skip Lookup in Update and Delete mode . . . 83
Using the Connector with ITIM Server 83
HTTPS (SSL) Support 84
Configuration 84
See also. 84
DSMLv2 SOAP Connector 85
Supported Connector Modes 85
Extended Operations 86
SOAPAction Header 86
Configuration 86
DSMLv2 SOAP Server Connector 89
Extended operations 89
Configuration 89
EIF Connector 91
Introduction to IBM Tivoli Netcool/OMNIbus . . 91
Introduction to Tivoli Enterprise Console . . . 91
Introduction to the Event Integration Facility . . 92
Schema 92
Configuration 93
File system Connector 95
Configuration 95
See also. 95
Form Entry Connector 97
Using the Connector 97
Configuration 97
FTP Client Connector 99
SSL support 99
Character Encoding 100
Configuration 100
See also 101
GLA Connector. 103
Introduction 103
Configuration 103
Configuring the TDIOutputter. 103
Using the Connector 104
Schema 104
See also 105
HTTP Client Connector 107
Modes. 107
Special attributes 108

Character Encoding 108
Configuration 109
Examples 109
See also 110
Old HTTP Client Connector 111
Modes 111
Special attributes 111
Configuration 112
Examples 112
See also 113
HTTP Server Connector 115
Connector structure and workflow 115
Connector Client Authentication 116
Chunked Transfer Encoding 116
Configuration 117
Connector Schema 118
See also 119
Old HTTP Server Connector 121
Configuration 121
See also 121
IBM Tivoli Directory Server Changelog Connector 123
Attribute merge behavior 123
Differences between changelog on distributed TDS and z/OS TDS 124
Configuration 124
See also 126
ITIM Agent Connector 129
Setting up SSL for the ITIM Agent Connector 129
Configuration 129
Known Issues 130
See also 130
IBM MQ Connector 131
JDBC Connector 133
Connector structure and workflow 133
Understanding JDBC Drivers 133
Specifying ODBC database paths 138
Schema 139
Configuration 139
Customizing select, insert, update and delete statements 142
Custom Prepared Statements 144
Additional JDBC Connector functions 146
Timestamps 147
Padding 148
Calling Stored Procedures 148
SQL Databases: column names with special characters 148
Using prepared statements 149
On Multiple Entries 149
Additional built-in reconnect rules 149
See also 149
JMS Connector 151
Introduction 151
JMS message flow 151
WebSphere MQ and JMS/non-JMS consumers of messages 152
JMS message types 152
Iterator mode 154
Lookup mode 154
AddOnly mode. 154
Call/Reply mode 154
JMS headers and properties 155
Configuration 156
Examples 158
External System Configuration 158
Troubleshooting 162
JMS Password Store Connector 165
Connector Workflow 165
Force transfer of accumulated messages from the JMS Password Store with MQe 166
PKCS7 Encryption support 166
Schema 169
Configuration 169
JMS drivers 171
See also 172
JMX Connector 173
Connector Schema 173
Configuration 174
See also 175
JNDI Connector 177
Configuration 177
Setting the Modify operation 178
Skip Lookup in Update and Delete mode . . . 180
See also 180
LDAP Connector 181
Detect and handle modrdn operation 182
Configuration 182
Virtual List View Control 185
Handling memory problems in the LDAP Connector 185
Built-in rules for reconnect functionality . . . 186
Searching against an SDBM backend on z/OS 186
LDAP Connector methods (API) 187
See also 189
LDAP Server Connector 191
Scripting 191
Returning the LDAP message returned values 191
Error handling 191
Configuration 192
See also 192
Log Connector 193
Introduction 193
Schema 193
Configuration 193
Lotus Notes Connector 201
Mailbox Connector 203
Configuration 203
Schema 204
Using the Connector 206
Memory Queue Connector 209
Memory queue components 209
High level workflow 210
Configuration 210
Accessing the Memory Queue programmatically 211
Memory Stream Connector 213
Configuration 213
Sun Directory Change Detection Connector . . . 215
Attribute merge behavior 215
Configuration 216
See also 218
Properties Connector 219
Configuration 219

Using the Connector 220
Properties File Format 221
See also 221
Server Notifications Connector 223
Encryption 223
Authentication 224
Configuration 224
Schema 225
System Queue Connector 227
Introduction 227
Configuration 227
Security, Authentication and Authorization . . 228
Windows Users and Groups Connector 231
Preconditions 231
Character sets 234
Examples 234
Windows Users and Groups Connector functional specifications and software requirements 234
System Store Connector 237
Configuration 237
Using the Connector 239
See also 240
RAC Connector 241
Introduction 241
Configuration 242
Using the Connector 242
See also 244
RDBMS Change Detection Connector 245
Configuration 245
Change table format 247
Creating change tables in DB2 247
Creating change tables in Oracle 248
Creating change table and triggers in MS SQL 249
Creating change table and triggers in Informix 251
Creating change table and triggers for SYBASE 252
Example 254
Script Connector 255
Predefined script objects 255
Functions 256
Configuration 257
Examples 258
See also 258
SNMP Connector 259
Configuration 259
Examples 259
See also 259
SNMP Server Connector 261
Connector Schema 261
Configuration 262
Tivoli Access Manager (TAM) Connector 263
Introduction 263
Connector Modes 263
Skip Lookup in Update and Delete mode . . . 263
Configuration 263
Using the Connector 266
Troubleshooting 272
Connector Input Attribute Details 273
See also 277
TCP Connector 279
Iterator Mode 279
AddOnly Mode 279
Configuration 279
See also 280
TCP Server Connector 281
Configuration 281
Connector Schema 281
See also 282
Timer Connector 283
Configuration 283
URL Connector 285
Configuration 285
Supported URL protocol 285
See also 285
Web Service Receiver Server Connector 287
Hosting a WSDL file 287
Schema 288
Configuration 288
Connector Operation 290
See also 290
z/OS LDAP Changelog Connector 291
Attribute merge behavior 291
Configuration 291
See also 293

Chapter 3. Parsers 295
Base Parsers 295
Character Encoding conversion 295
Availability 296
CBE Parser 297
Using the Parser 297
CBE Input and Output Map Attributes 297
Configuration 300
See also 300
CSV Parser 301
Configuration 301
Schema 302
DSML Parser 303
Configuration 303
Examples 303
See also 304
DSMLv2 Parser 305
Modes 305
Operations 305
Binary and non-String Attributes 311
Optional Attributes 312
DSMLv2 controls must be Base64 encoded . . 312
Setting result code and result description . . . 312
Multiple Attribute modifications 312
Configuration 313
Examples 314
Fixed Parser 317
Configuration 317
HTTP Parser 319
Configuration 319
Schema 319
Character sets/Encoding 324
How to use HTTP cookies 324
See also 324
LDIF Parser 325
Reading LDIF input 325
Writing LDIF output 325

Configuration 326
See also 327
Line Reader Parser 329
Configuration 329
Script Parser 331
Objects 331
Functions (methods) 332
Configuration 333
Schema 333
Example 333
See also 334
Simple Parser 335
Configuration 335
SOAP Parser 337
Example Entry 337
Example SOAP document 337
Configuration 337
Parser-specific calls 338
Examples 338
SPMLv2 Parser 339
Introduction 339
Operations 339
Configuration 345
Example 346
See also 346
Simple XML Parser 347
Configuration 347
Character Encoding in the Simple XML Parser 348
Examples 348
Additional Examples 350
See also 350
XML Parser 351
Introduction 351
Configuration 351
Using the Parser 352
Example 358
Using XSD Schemas 358
XML SAX Parser 361
Configuration 362
Character encoding 362
See also 363
XSL based XML Parser 365
Introduction 365
Configuration 365
Using the Parser 366
See also 367
User-defined parsers 369

Chapter 4. Function Components . . . 371

Castor Java to XML Function Component 373
Castor Overview 373
Configuration 373
Using the FC 374
Castor XML to Java Function Component 375
Configuration 375
Using the FC 376
XMLToSDO Function Component 377
Example 377
Configuration 378
Migration 378
SDOToXML Function Component 381
Configuration 382
Using the FC 382
Migration 382
AssemblyLine Function Component 385
Configuration 385
Using the FC 386
See also 387
Java Class Function Component 389
Schema 389
Configuration 389
Parser Function Component 391
Configuration 391
Using the FC 391
Scripted Function Component 393
Configuration 393
Using the FC 393
Objects 393
See also 394
CBE Function Component 395
Common Base Event (CBE) 395
The Common Event Infrastructure (CEI) . . . 395
Input and Output attributes 396
Configuration 396
Generating a CBE Log XML 396
Emitting events to a CEI Server 397
Function Component API 397
See also 398
SendEMail Function Component 399
Schema 399
Configuration 400
Memory Queue Function Component 402
Configuration 402
Using the FC 402
See also 403
Axis Java To Soap Function Component 405
Configuration 405
Using the FC 406
WrapSoap Function Component 409
Configuration 409
Using the FC 410
InvokeSoap WS Function Component 411
Introduction 411
Authentication 411
Configuration 411
Using the FC 412
See also 413
Axis Soap To Java Function Component 415
Configuration 415
Using the FC 415
Axis2 WS Client Function Component 417
Using the FC 417
Supported Message Exchange Patterns 417
SOAP Headers 417
Schema 417
Configuration 420
See also 420
Axis EasyInvoke Soap WS Function Component 421
Configuration 421
Using the FC 422
See also 423
Complex Types Generator Function Component 425

Configuration 425
Function Component Input and Output . . . 425
Troubleshooting 426
Delta Function Component 427
Introduction 427
Configuration 427
Using the Function Component 428
Example 428
Remote Command Line Function Component . . 431
Configuration 431
Function Component Input 432
Function Component Output 433
Using the FC 433
See also 435
z/OS TSO/E Command Line Function Component 437
Configuration 437
Using the FC 437
Required pseudonym file 439
Setting up the native part of the FC 443
See also 444

Chapter 5. SAP ABAP Application Server Component Suite 445
Who should read this chapter 445
Component Suite Installation 445
Software Requirements 445
Verifying the Component Suite for SAP ABAP Application Server 446
Checking the Version Numbers 447
Uninstallation 448
Function Component For SAP ABAP Application Server 449
Function Component Introduction 449
Configuration 449
Using the Function Component 451
User Registry Connector for SAP ABAP Application Server 453
Introduction 453
Skip Lookup in Update and Delete mode . . . 454
Configuration 454
Using the User Registry Connector for SAP ABAP Application Server 457
Human Resources/Business Object Repository Connector for SAP ABAP Application Server . . . 461
Introduction 461
Skip Lookup in Update and Delete mode . . . 463
Configuration 464
Using the Human Resources Connector for SAP ABAP Application Server 466
ALE Intermediate Document (IDOC) Connector for SAP ABAP Application Server and SAP ERP . . . 471
Introduction 471
Installation 472
Configuration 472
Using the SAP ALE IDOC Connector 474
Troubleshooting the SAP ABAP Application Server Component Suite 483
Supplemental information for the SAP ABAP Application Server Component Suite 485
Example User Registry Connector XML Instance Document 485
XSchema for User Registry Connector XML . . 487

Chapter 6. Asset Integration Suite . . 497
Overview 497
CDM components 497
IT registry 499
Components of the suite 500
Open IdML Function Component 500
Close IdML Function Component 504
Rolling IdML Function Component 504
IdML CI and Relationship Connector 506
IdML Parser 508
Data Cleanser Function Component 510
Init IT Registry Function Component 510
IT Registry CI and Relationship Connector . . 512
The it_registry.properties file 516
Examples 516
IT Registry database setup 517
Troubleshooting 518

Chapter 7. Script languages 521
JavaScript 521
Java and JavaScript 521

Chapter 8. Objects 523
The AssemblyLine Connector object 523
The attribute object 523
Examples 523
See also 524
The Connector Interface object 524
Methods 524
The Entry object 524
Global Entry instances available in scripting . . 525
See also 526
The FTP object 526
Example 526
Main object 526
The Search (criteria) object 527
Operands 527
Example 527
The shellCommand object 527
The status object 527
The system object 527
The task object 527
The COMProxy object 528
Example code 528
See also 530

Appendix A. Password Synchronization plug-ins 531

Appendix B. AssemblyLine Flow Diagrams 533
AssemblyLine Flow Diagrams 533

Appendix C. Server API 535
Overview 535
Sample use case 536
Local and Remote Server API interfaces 536

Server API structure 537
Security 537
Configuring the Server API 538
Configuring the Server API properties 538
Setting up the User Registry 538
Remote client configuration 538
Using the Server API 540
Creating a local Session 540
Creating a remote Session 540
Working with Config Instances 541
Working with AssemblyLines 544
Editing configurations 547
Working with the System Queue 551
Working with the Tombstone Manager 552
Working with TDI Properties 555
Registering for Server API event notifications 555
Getting access to log files 557
Server Info 558
Using the Security Registry 559
Custom Method Invocation 559
The JMX layer 561
Local access to the JMX layer 561
Remote access to the JMX layer 561
MBeans and Server API objects 562
JMX notifications 562
JMX Example - Tivoli Directory Integrator 7.1 and MC4J configuration 563
Backward compatibility 566
Scenarios overview 566
Server API changes in Tivoli Directory Integrator 7.1 569
Known issues 573

Appendix D. Creating new components using Adapters 575
Introduction 575
Features that enable implementation of a Tivoli Directory Integrator Adapter 576
AL Operations 576
Switch/case component 576
Flexible connector initialization 577
Using an Iterator in Flow 577
Packing an Adapter for consumption 577
Using an Adapter in your AssemblyLine . . . 578
The use of operations in a Tivoli Directory Integrator Adapter 578
Mapping Adapter operations to Connector modes 578
Implementing code in the Adapter for each operation 579
Adapter configuration through the $initialization operation 579
Understanding the link criteria 580
Attribute mapping 580
Status indication 581
Implementing Query Schema 581
Delta mode 581
Error handling 582

Appendix E. Implementing your own Components in Java 583
Support materials for Component development 583
Developing a Connector 583
Implementing the Connector's Java source code 583
Building the Connector's source code 591
Implementing the Connector's GUI configuration form 591
Connector Reconnect Rules definition 599
Packaging and deploying the Connector . . . 600
Developing a Function Component 600
Implementing Function Component Java source code 600
Building the Function Component source code 601
Implementing the Function Component GUI configuration form 601
Packaging and deploying the Function Component 601
Developing a Parser 601
Implementing the Parser Java source code . . . 601
Building the Parser source code 603
Implementing the Parser GUI configuration form 603
Packaging and deploying the Parser 603
Creating additional Loggers 604
Understanding the logging interface 605
See also 611

Appendix F. Notices 613
Third-Party Statements 614
ICU License - ICU 1.8.1 and later 614
Trademarks 615

Chapter 1. Introduction

To work with examples complementing this manual, you must refer back to the installation package to download the necessary files.

To access these example files, go to the *root_directory*/examples directory in the installation directories.

Chapter 2. Connectors

Connector availability and reference

The following is a list of all Connector Interfaces included with the IBM Tivoli Directory Integrator. The Connector Interface is the part of the Connector that implements the actual logic to communicate with the Data Source it is supposed to handle.

You can also make your own Connector Interfaces if needed; the AssemblyLine wraps them so they are available as AssemblyLine Connectors.

Before Connectors can be meaningfully deployed in an AssemblyLine, it needs to be configured. A number of Connectors have different parameter sets, depending on the Mode they are set to; this implies that, for example, a parameter which is significant in Iterator mode, is not necessary and therefore not present in the list of parameters in AddOnly Mode.

All following AssemblyLine Connectors have access to the methods described in the `com.ibm.di.server.`*`AssemblyLineComponent`* in addition to the methods and properties of the Connector Interface. For documentation of the methods, see the JavaDocs (from the CE, choose **Help -> Welcome -> JavaDocs**.)

Connector Interfaces

For a list of Supported Modes, see "Legend for the Supported Mode columns" on page 6.

For each Connector Interface listed, see the documentation outlined in this chapter.

"Active Directory Change Detection Connector" on page 8
I

"AssemblyLine Connector" on page 14
I

"Axis Easy Web Service Server Connector" on page 19
S

"Axis2 Web Service Server Connector" on page 25
S

"Command line Connector" on page 37
A I C

"Direct TCP /URL scripting" on page 41
custom

"Domino AdminP Connector" on page 73
I A

"Domino Change Detection Connector" on page 49
I

"Domino Users Connector" on page 59
A D I L U

"DSMLv2 SOAP Connector" on page 85
A D I L U C Δ

"DSMLv2 SOAP Server Connector" on page 89
S

“EIF Connector” on page 91
A I

“File system Connector” on page 95
A I

“Form Entry Connector” on page 97
I

“FTP Client Connector” on page 99
A I

“GLA Connector” on page 103
I

“Old HTTP Client Connector” on page 111
A D I L U C Δ S

“HTTP Client Connector” on page 107
A I L C

“Old HTTP Server Connector” on page 121
A I

“HTTP Server Connector” on page 115
I S

“Human Resources/Business Object Repository Connector for SAP ABAP Application Server” on page 461 A D I L U

“IBM Tivoli Directory Server Changelog Connector” on page 123
I

“IBM MQ Connector” on page 131
A I L C

“JDBC Connector” on page 133
A D I L U Δ

“JMS Connector” on page 151
A I L C

“JMS Password Store Connector” on page 165
I

“JMX Connector” on page 173
I

“JNDI Connector” on page 177
A D I L U Δ

“LDAP Connector” on page 181
A D I L U Δ

“LDAP Server Connector” on page 191
S

“Log Connector” on page 193
A

“Lotus Notes Connector” on page 77
A D I L U

“Mailbox Connector” on page 203
A D I L U

“Memory Queue Connector” on page 209
A I

“Memory Stream Connector” on page 213
A I

“Sun Directory Change Detection Connector” on page 215
I

“System Store Connector” on page 237
A D I L U

“RAC Connector” on page 241
A I

“RDBMS Change Detection Connector” on page 245
I

“Script Connector” on page 255
custom

You write the Script Connector yourself, and it provides the modes you write into it.

“Server Notifications Connector” on page 223
A I

“SNMP Connector” on page 259
A I L

“SNMP Server Connector” on page 261
S

“Properties Connector” on page 219
A I U L D

“System Queue Connector” on page 227
A I

“Tivoli Access Manager (TAM) Connector” on page 263
A I D L U

“TCP Connector” on page 279
A I

“TCP Server Connector” on page 281
I S

“ITIM Agent Connector” on page 129
A D I L U

“ITIM DSMLv2 Connector” on page 83
A D I L U

“Timer Connector” on page 283
I

“URL Connector” on page 285
A I

“User Registry Connector for SAP ABAP Application Server” on page 453
A D I L U

“Web Service Receiver Server Connector” on page 287
S

“Windows Users and Groups Connector” on page 231
A D I L U

“z/OS LDAP Changelog Connector” on page 291
I

Legend for the Supported Mode columns

- A–AddOnly
- D–Delete
- I–Iterator
- L–Lookup
- U–Update
- Δ–Delta
- C–Call/Reply
- S–Server
- +–Newer version support exists

Script-based Connectors

For a list of Supported Modes, see “Legend for the Supported Mode columns.” The Script Connector enables you to write your own Connector in JavaScript™.

Generic Connector
custom

You write the Script Connector yourself in JavaScript, and it provides the modes you write into it. See "JavaScript Connector" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.

In Script-based Connectors, a potential source of problems exists if you made direct Java calls into the same libraries as IBM Tivoli Directory Integrator. A new version of IBM Tivoli Directory Integrator might have updated libraries (with different semantics), or you might have upgraded your libraries since the last time you used your Connector.

Configurations

For a list of Supported Modes, see “Legend for the Supported Mode columns.”

Connector re-use

When a Connector is instantiated, usually it allocates a certain amount of resources to communicate with a particular system (connection objects, session objects, result set, and so forth). When multiple Connectors of the same type are connected to the same system, often it is reasonable to share the underlying resources. This means that a single connection to the given system will be re-used by multiple Connectors.

Tivoli Directory Integrator allows Connector re-use to happen within an AssemblyLine. For a given AssemblyLine the you have the option to re-use an already configured Connector from the same AssemblyLine.

With regards to the Tivoli Directory Integrator Server, when re-using a Connector, a single physical Connector object is instantiated and a number of logical Connectors share it.

With regards to configuration, Connector re-using is a master-slave relation: the re-used ("master") Connector has a full connection and parser configuration and all re-using Connectors have references to the master Connector. All re-using Connectors share the connection and parser settings of the Connector they re-use. Although connection and parser settings are fixed for re-using Connectors, certain other features are configured separately (if any parameter is not configured separately, it is inherited from the master Connector):

- Input/Output Map

- Link Criteria
- Hooks
- Delta settings
- Reconnect settings

Generally, a Connector can be re-used in the same mode (except for Iterator and Server) without any problem. This means that, for example, you can safely re-use a Connector in Lookup mode as many times as you wish.

A problem can potentially arise when a Connector is re-used in different modes. The shared physical Connector object is initialized and terminated only once. So the Connector's initialization and termination procedure must be common for all supported modes.

Following is a list of Tivoli Directory Integrator Connectors which can be re-used in different modes:
- Domino Users Connector
- DSMLv2 SOAP Connector
- HTTP Client Connector
- IBM MQ Connector
- ITIM Agent Connector
- JDBC Connector
- JMS Connector
- JNDI Connector
- LDAP Connector
- Lotus Notes Connector
- Mailbox Connector
- Properties Connector
- SAP ABAP Application Server Business Object Repository Connector
- SAP ABAP Application Server User Registry Connector
- Script Connector (depends on the user-supplied Javascript)
- SNMP Connector
- System Queue Connector
- System Store Connector
- TAM Connector
- TCP Connector
- ITIM DSMLv2 Connector
- URL Connector
- Windows Users and Groups Connector

Any Connector not in this list can not be re-used in the same AssemblyLine; either because it makes no sense, or because the Connector's internal logic does not allow it.

For configuring a Connector for re-use in an AssemblyLine, refer to *IBM Tivoli Directory Integrator V7.1 Users Guide*. In the configured AssemblyLine, the re-used Connectors will show up with their name prepended with '@'.

Active Directory Change Detection Connector

The Active Directory Change Detection Connector (hereafter referred to as ADCD Connector) is a specialized instance of the LDAP Connector. It reports changed Active Directory objects so that other repositories can be synchronized with Active Directory.

The LDAP protocol is used for retrieving changed objects.

When run the Connector reports the object changes necessary to synchronize other repositories with Active Directory regardless of whether these changes occurred while the Connector has been offline or they are happening as the Connector is online and operating.

This connector also supports Delta Tagging, at the Entry level only.

The ADCD Connector operates in Iterator mode.

Note: This component is not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.

Tracking changes in Active Directory

Active Directory does not provide a Changelog as IBM Directory Server and some other LDAP Servers do.

The ADCD Connector uses the **uSNChanged** Active Directory attribute to detect changed objects.

Each Active Directory object has an **uSNChanged** attribute that corresponds to a directory-global USN (Update Sequence Number) object. Whenever an Active Directory object is created, modified or deleted, the global sequence object value is increased, and the new value is assigned to the object's **uSNChanged** attribute.

On each AssemblyLine iteration (each call of the `getNextEntry()` Connector's method) it delivers a single object that has changed in Active Directory. It delivers the changed Active Directory objects as they are, with all their current attributes and also reports the type of object change – whether the object was updated (added or modified) or deleted. The Connector does not report which attributes have changed in this object and the type of attribute change.

Synchronization state is kept by the Connector and saved in the User Property Store – after each reported changed object the Connector saves the USN number necessary to continue from the correct place in case of interruption and restart; when started, the ADCD Connector reads this USN value from the IBM Tivoli Directory Integrator's User Property Store stored from the most recent ADCD Connector session.

Information from MSDN about tracking changes in Active Directory can be found here, and information about polling for changes using the uSNChanged attribute is here.

Deleted objects in Active Directory

When an object is deleted from the directory, Active Directory performs the following steps:

- The object's **isDeleted** attribute is set to TRUE. Objects where isDeleted==TRUE are known as tombstones (not related to TDI tombstones).
- All attributes that are not needed by Active Directory are removed. A few key attributes, including **objectGUID**, **objectSID**, **nTSecurityDescriptor**, and **uSNChanged** are preserved.
- Moves the tombstone to the Deleted Objects container, which is a hidden container within the directory partition.

Tombstones or deleted objects are garbage collected some time after the deletion takes place. Two settings on the "cn=Directory Service,cn=Windows NT,cn=Service,cn=Configuration,dc=ForestRootDomain" object determine when and which tombstones are deleted:

- The "garbage collection interval" determines the number of hours between garbage collection on a domain controller. The default setting is 12 hours, and the minimum setting is 1 hour.
- The "tombstone lifetime" determines the number of days that tombstones persist before they are vulnerable to garbage collection. The default setting is 60 days, and the minimum setting is 2 days.

The above specifics imply the following requirements for synchronization processes that have to handle deleted objects:

- Synchronization has to be run on intervals shorter than the "tombstone lifetime" Active Directory setting.
- The **objectGUID** attribute has to be used for object identifier during synchronization. The object's **distinguishedName** attribute which uniquely identifies the position of an object in the directory tree, cannot be used because after the object is deleted it changes its place in the directory tree – it is moved in the Deleted Objects container and its old distinguished name is irrevocably lost. The **objectGUID** attribute is however never changed. When a deleted object is found during synchronization, a search in the other repository for an object with the same **objectGUID** should be made and the found object should be deleted.

Moved objects in Active Directory

When an object is moved from one location of the Active Directory tree to another, its **distinguishedName** attribute changes. When this object change is detected based on the new increased value of the object's **uSNChanged** attribute, this change looks like any other modify operation - there is no information about the object's old distinguished name.

A synchronization process that has to handle moved objects properly should use the **objectGUID** attribute – it doesn't change when objects are moved. A search by the **objectGUID** attribute in the repository which is synchronized will locate the proper object and then the old and new distinguished names can be compared to check if the object has been moved.

Use objectGUID as the object identifier

When tracking changes in Active Directory the **objectGUID** attribute should be used for object identifier and not the LDAP distinguished name. This is so because the distinguished name is lost when an object is deleted or moved in Active Directory. The **objectGUID** attribute is always preserved, it never changes and can be used to identify an object.

When the ADCD Connector reports that an entry is changed, a search by **objectGUID** value should be performed in the other repository to locate the object that has to be modified or deleted. This means that the **objectGUID** attribute should be synchronized and stored into the other repository.

Change detection

Change detection mechanism

The ADCD Connector detects and reports changed objects following the chronology of the **uSNChanged** attribute values: changed objects with lower **uSNChanged** values will be reported before changed objects with higher **uSNChanged** values.

The Connector executes an LDAP query of type `(usnChanged>=X)` where X is the USN number that represents the current synchronization state. Sort and Page LDAP v3 controls are used with the search operation and provide for chronology of changes and ability to process large result sets. The Show Deleted LDAP v3 request control (OID "1.2.840.113556.1.4.417") is used to specify that search results should include deleted objects as well.

The ADCD Connector consecutively reports all changed objects regardless of interruptions, regardless of when it is started and stopped and whether the changes happened while the Connector was online or

offline. Synchronization state is kept by the Connector and saved in the User Property Store – after each reported changed object the Connector saves the USN number necessary to continue from the correct place in case of interruption and restart.

The Connector will signal end of data and stop (according to the timeout value) when there are no more changes to report.

When there are no more changed Active Directory objects to retrieve, the Active Directory Connector cycles, waiting for a new object change in Active Directory. The **Sleep Interval** parameter specifies the number of seconds between two successive polls when the Connector waits for new changes. The Connector loops until a new Active Directory object is retrieved or the timeout (specified by the **Timeout** parameter) expires. If the timeout expires, the Active Directory Connector returns a **null** Entry, indicating there are no more Entries to return. If a new Active Directory object is retrieved, it is processed as previously described, and the new Entry is returned by the Active Directory Connector.

In older versions of the Connector, it reported both added and modified entries as updated. Currently, the Connector differentiates between add and modify and reports each operation separately (for details see “Offline and Paged results cases.”)

The ADCD Connector delivers changed Active Directory objects as they are, with all their current attributes. It does not determine which object attributes have changed, nor how many times an object has been modified. All intermediate changes to an object are irrevocably lost. Each object reported by the Active Directory Connector represents the cumulative effect of all changes performed to that object. The Active Directory Connector, however, recognizes the type of object change that has to be performed on the replicated data source and reports whether the object must be updated or deleted in the replicated data source.

Note: You can retrieve only objects and attributes that you have permission to read. The Connector does not retrieve an object or an attribute that you do not have permission to read, even if it exists in Active Directory. In such a case the ADCD Connector acts as if the object or the attribute does not exist in Active Directory.

Offline and Paged results cases

When the Connector is offline or when **Paged results** is enabled and an initial search request is made but the page containing modified entry is not retrieved yet, multiple changes made to that entry are merged. In other words the Connector receives only one entry containing the results of all operations that have been applied on it.

In these cases when an entry is **added** and then **deleted** in Active Directory the Connector will report "**delete**" operation for entries that have not been added to the repository being synchronized with Active Directory. This is not a serious restriction because IBM Tivoli Directory Integrator 7.1's Delete Connector mode first checks if the entry to be deleted exists and if it does not exist, the "On No Match" hook is called - this is where you can place code to handle/ignore such unnecessary deletes.

Another scenario is when entry is **added** and then **modified**. In that case the Connector will report an "**add**" operation for that entry, and the entry will contain all the changes made to it after the adding.

In all other possible cases the return entry will contain all the changes and it will be tagged with the last operation made to it.

Using the Active Directory Change Detection Connector

Each delivered entry by the Connector contains the **changeType** attribute whose value is either "add" (for newly created objects), "modify" (for modified objects) or "delete" (for deleted Active Directory objects). Each entry also contains 2 attributes that represent the objectGUID value:

- attribute **objectGUID** – contains a 16-byte byte array that represents the 128-bit objectGUID of the corresponding Active Directory object.
- attribute **objectGUIDStr** – contains the string representation of the hexadecimal value of the 128-bit objectGUID. It is delivered in the format {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}, where each x represents a hexadecimal digit.

If you need to detect and handle moved or deleted objects, you must use the **objectGUID** value as object identifier instead of the LDAP distinguished name. The LDAP distinguished name changes when an object is moved or deleted, while the **objectGUID** attribute always remains unchanged. Store the objects' **objectGUID** attribute in the replicated data source and search by this attribute to locate objects.

Note: Deleted objects in Active Directory live for a configurable period of time (60 days by default), after which they are completely removed. To avoid missing deletions, perform incremental synchronizations more frequently.

The ADCD Connector can be interrupted at any time during the synchronization process. It saves the state of the synchronization process in the User Property Store of the IBM Tivoli Directory Integrator (after each Entry retrieval), and the next time the Active Directory Connector is started, it successfully continues the synchronization from the point the Active Directory Connector was interrupted.

Authentication of the Connector to the directory

Different versions of the LDAP protocol support different authentication methods. LDAP v2 supports three: Anonymous, Simple, Kerberos v4. LDAP v3 supports: Anonymous, Simple and SASL authentication. The AD Change Detection Connector supports Anonymous, Simple and SASL authentication.

Anonymous

If this authentication method is set, it means that the server, to which a client is connected, does not know or care who the client is. The server allows such client to access data that is configured for non-authenticated users. The ADCD connector automatically specifies this authentication method if no username is supplied. If this type of authentication is chosen and a username is specified, then the ADCD connector automatically sets the authentication method to Simple.

Simple

In this case the LDAP server requires that the client (ADCD Connector) sends the fully qualified distinguished name and the client password in cleartext. This is a problem, because the password may be read from the network. You may use this type of authentication in combination with the SSL protocol to avoid exposing the password, if the LDAP server supports it. If Simple mode is specified and neither a username, then mode is automatically set to Anonymous by the ADCD Connector. If Anonymous mode is chosen, but a username is specified, then the mode is set to Simple by the ADCD Connector.

SASL The client (the ADCD Connector) will use a Simple Authentication and Security Layer (SASL) authentication method when connecting to the LDAP Server. In order to use this authentication mechanism, you need to configure the SASL mechanism to be used manually by specifying additional JNDI parameters using the **Extra Provider Parameters** option. For more information on SASL authentication, the SASL mechanisms supported by JNDI, and configurable SASL parameters available to JNDI, refer to the following URL: http://java.sun.com/products/jndi/tutorial/ldap/security/sasl.html.

Note: Not all directory servers support all SASL mechanisms and in some cases do not have them enabled by default. Check the documentation and configuration options for the directory server you are connecting to for this information.

Here is an example of the parameters to add to the **Extra Provider Parameters** parameter to configure the LDAP Connector to use DIGEST-MD5 when the SASL authentication method is selected:

```
java.naming.security.authentication:DIGEST-MD5
```

Error flows

- A PartialResultException is thrown if the returned entry contains referrals – an Active Directory server is able to generate referrals to external domains. However it is the client's responsibility to be able to follow these referrals because Active Directory does not perform that; it does not support the Manage Referral control. Therefore, the Active Directory Change Detection Connector cannot follow referrals either.
- An Exception is thrown if the **LDAP URL** parameter is missing.
- The Connector will throw an exception when there is a problem in storing the USN value in the System Store.

Configuration

The Connector needs the following parameters:

LDAP URL
Specifies the LDAP URL of the Active Directory service you want to access. The LDAP URL has the form ldap://*hostname*:*port* or ldap://*server_IP_address*:*port*. For example, **ldap://localhost:389**

Note: The default LDAP port number is 389. When using SSL, the default LDAP port number is 636.

Login username
Specifies the distinguished name used for authentication to the service. For example, **cn=administrator,cn=users,dc=your_domain,dc=com**.

Note: If you use Anonymous authentication, you must leave this parameter blank.

Login password
Specifies the credentials (password).

Note: If you use Anonymous authentication, you must leave this parameter blank.

Authentication Method
Specifies the authentication method to be used. Possible values are:

- Anonymous (use no authentication)
- Simple (use weak authentication (cleartext password))

Use SSL
Specifies whether to use Secure Sockets Layer for LDAP communication with Active Directory.

Extra Provider Parameters
Allows you to pass a number of extra parameters to the JNDI layer. It is specified as name:value pairs, one pair per line.

Binary Attributes
Specifies a list of parameters that are to be interpreted as binary values instead of strings. The default value for this parameter is **objectGUID objectSid**.

LDAP Search Base
Specifies the Active Directory sub-tree that is polled for changes. The search base should be an Active Directory Naming Context if detection of deleted objects is required. For example, **dc=your_domain,dc=com**.

Page Size
Specifies the number of entries per page returned by this request (default value is 500).

Iterator State Key
Specifies the name of the parameter that stores the current synchronization state in the User Property Store of the IBM Tivoli Directory Integrator. This must be a unique name for all

parameters stored in one instance of the IBM Tivoli Directory Integrator User Property Store. The **Delete** button lets you delete this information from the User Property Store.

Start at
Specifies either **EOD** or **0**. **EOD** means report only changes that occur after the Connector is started. **0** means perform full synchronization, that is, report all objects available in Active Directory Service. This parameter is taken into account only when the parameter specified by the **Iterator State Key** parameter is not found in the User Property Store.

State Key Persistence
Determines when the Connector's state is written to the System Store. The default (and recommended setting) is **End of Cycle**, and the choices are:

After read
Updates the System Store when you read an entry from the Active Directory change log, before you continue with the rest of the AssemblyLine.

End of cycle
Updates the System Store with the change log number when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual
Switches off the automatic updating of the System Store with this Connector's state information; instead, you will need to save the state by manually calling the ADCD Connector's *saveStateKey()* method, somewhere in your AssemblyLine.

Use Notifications
Specifies whether to use notification when waiting for new changes in Active Directory. If not enabled, the Connector will poll for new changes.

If enabled, the Connector will not sleep or timeout but instead wait for a Change Notification event (*Server Search Notification Control* (OID 1.2.840.113556.1.4.528) from the Active Directory server, and the sleep interval and timeout parameters are ignored.

Timeout
Specifies the maximum number of seconds the Connector waits for the next changed Active Directory object. If this parameter is **0**, then the Connector waits forever. If the Connector has not retrieved the next changed Active Directory object within *timeout* seconds, then it returns an empty (**null**) Entry, indicating that there are no more Entries to return. The default is 5.

Sleep Interval
Specifies the number of seconds the Connector sleeps between successive polls.

Detailed Log
If this field is checked, additional log messages are generated.

Comment
Your comments here.

Note: Changing Timeout or Sleep Interval values will automatically adjust its peer to a valid value after being changed (for example, when timeout is greater than sleep interval the value that was not edited is adjusted to be in line with the other). Adjustment is done when the field editor looses focus.

See also

"LDAP Connector" on page 181,
"Sun Directory Change Detection Connector" on page 215,
"IBM Tivoli Directory Server Changelog Connector" on page 123
"z/OS LDAP Changelog Connector" on page 291
How to poll for object attribute changes in Active Directory on Windows 2000 and Windows Server 2003.

AssemblyLine Connector

AssemblyLines are often called as compound functions from other AssemblyLines. Setting up a call to perform a specific task and mapping in and out parameters can be tedious in a scripting environment. To ease the integration of AssemblyLines into a work flow, the AssemblyLine Connector provides a standard and familiar way of doing this; it wraps much of the scripting involved to execute an AssemblyLine. The AssemblyLine connector uses the AssemblyLine manual cycle mode for inline execution; and internally it uses the “AssemblyLine Function Component” on page 385 to do its work.

The AssemblyLine Connector supports Iterator mode only, except when calling another AssemblyLine which supports AssemblyLine Operations. See "AssemblyLine Operations" in *IBM Tivoli Directory Integrator V7.1 Users Guide* and Appendix D, “Creating new components using Adapters,” on page 575 for more information.

The server-server capability made possible by using this Connector addresses security concerns when managers want Tivoli Directory Integrator developers to access connected systems, but not to access the operational parameters of the Connector – or to impact its availability by deploying the new function on the same physical server.

Configuration

The Connector needs the following parameters:

AssemblyLine
: The name of the AssemblyLine to be executed under control of this Connector. Select from the drop-down list or enter the name.

AssemblyLine Parameters
: The AssemblyLine parameters as defined in the target AssemblyLine AL Operations schema tab under *Published AssemblyLine Initialization Parameters*. It is not a real operation. The names defined in this schema will be used when configuring an AssemblyLine Connector or AL Function Component calling this AssemblyLine. The Information field for the name in the Published AssemblyLine Initialize Parameters schema will be used as tooltip (description) for the parameter.

  When the target AssemblyLine is created, all these attributes and values provided by you here are passed to the target AssemblyLine before any of the connectors are initialized. Access to these attribute/value pairs can be done through scripting as in:
  `task.getOpEntry().getAttribute("<name from schema>")` or by using the expression editor under the operations folder.

Remote Server
: The Tivoli Directory Integrator server on which the AssemblyLine is defined and will be run. Use blank for local instance or *host[:port]*.

Config Instance
: The ID or path of the Config instance on the remote server.

Custom Keystores
: Check this box to use the custom `api.remote.server.` java properties instead of standard `javax.net.ssl.` properties for keystore configuration.If you do so, the following properties from `global.properties` become relevant (see also "global.properties" in *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*):

  `api.client.keystore`
  : Specifies the keystore file containing the client certificate

  `api.client.keystore.pass`
  : Specifies the password of the keystore file specified by api.client.keystore

`api.client.key.pass`
The password of the private key stored in keystore file specified by `api.client.keystore`; if this property is missing, the password specified by `api.client.keystore.pass` is used instead.

`api.truststore`
Specifies the keystore file containing the Tivoli Directory Integrator Server public certificate.

`api.truststore.pass`
Specifies the password for the keystore file specified by `api.truststore`.

Simulate
If set, the called AssemblyLine will make use of its Simulation Config when interacting with external systems.

Share Logging
If set, the called AssemblyLine will use the same logging as this Connector. By default, this parameter is not set.

Detailed log
If this field is checked, additional log messages are generated.

Using the Connector

The AssemblyLine Connector iterates on the result set from the target AssemblyLine which is always run synchronously in manual cycle mode by the AssemblyLine Connector. The target AssemblyLine can be local to the thread or on a remote server by use of the Server API.

Note that most of the functionality is implemented in the "AssemblyLine Function Component" on page 385 component, so the AssemblyLine Connector simply redirects the occurring errors.

Attribute Mapping (Schema) and modes

The AssemblyLineConnector dynamically reports its available connector modes (for example, Iterator) based on the available operations in the target AssemblyLine. The target AssemblyLine can define any operation name which will appear in the connector's mode drop-down list. Any operation/mode that is not a standard mode name will implicitly use CallReply mode internally (that is, the UI changes to the CallReply equivalent layout and the queryReply method is invoked on the AssemblyLineConnector). To further aid in development of custom connectors, the AssemblyLine connector gives the operation names listed below special significance. The operation names are the same as the function names for the ScriptConnector and also the same names as the ConnectorInterface method names.

Table 1.

Computed Mode	Required Operations
Iterator	getNextEntry selectEntries
AddOnly	putEntry
Lookup	findEntry
Update	findEntry modEntry putEntry
Delete	findEntry deleteEntry
CallReply	queryReply

Table 1. (continued)

Computed Mode	Required Operations
N/A	initialize terminate *These two are optional operations but will be invoked if present.*

When one or more of these are present, the AssemblyLine Connector will compute supported modes based on the operations and the target AssemblyLine is said to be in adapter mode. The difference between normal mode and adapter mode is how the AssemblyLine connector calls the target AssemblyLine's operations.

As an example, if the target AssemblyLine implements findEntry as an operation, the UI will show Lookup as an available mode. When the AssemblyLine Connector is called by the AssemblyLine, it will forward the "native" methods (for example, findEntry) directly to the target AssemblyLine by invoking the findEntry operation. Another example is Delete where the AssemblyLine connector will invoke findEntry followed by deleteEntry to perform a delete operation. In normal mode (for example, the target AssemblyLine defines the DeleteUser operation), the AssemblyLine Connector would simply invoke the DeleteUser operation leaving the entire delete operation up to the target AssemblyLine. Although the target AssemblyLine can define standard modes as operations, this is not recommended as some operations will simply not function correctly because they require more than one operation to complete the mode operation (like delete and update that calls findEntry before deleting or updating).

When the AssemblyLine Connector invokes an operation in an adapter mode AssemblyLine it will pass the result from its output attribute map to the target AssemblyLine's work entry. In cases where a link criteria is required, the AssemblyLine Connector adds the `com.ibm.di.server.SearchCriteria` instance object to the op-entry of the target AssemblyLine as search. The target AssemblyLine can retrieve this object by calling `task.getOpEntry()..getObject("search")`.

The result from the target AssemblyLine is always communicated back in the work entry. This entry becomes the *conn* entry of the AssemblyLine Connector which is then subjected to its input attribute map. One exception to this rule is when the resulting work entry contains an attribute named "conn". When this attribute is present, the AssemblyLine connector will disregard all attributes in the returned work entry and use the *conn* attribute as the result from the operation. The *conn* entry can contain any number of Entry objects. This is typically used when findEntry returns either null or more than one entry. If the *conn* attribute has no values it is the equivalent of returning null, which will cause the *on-no-match* hook to be called. When the *conn* attribute has more than one value, the AssemblyLine Connector will add all entries to its multiple-found array so that the AssemblyLine triggers the *on multiple found* hook and makes the duplicate entries available using the `getFindEntryCount()` and `getNextFindEntry()` methods. If the *conn* attribute contains objects that are not of type `com.ibm.di.entry.Entry` an error will be thrown.

If the AssemblyLine Connector's target AssemblyLine has no operations defined the AssemblyLine Connector will report Iterator as its only mode. The AssemblyLine connector will not invoke operations on the target AssemblyLine but simply invoke the `executeCycle(work)` of the target AssemblyLine to get the next input entry.

Schema Discovery:

The schema for an AssemblyLine Connector can be retrieved after the Connector has been configured with the correct target AssemblyLine, and the mode to use has been chosen. In order to facilitate discovering the schema the following considerations apply:

1. The target AL is checked to see if it has an operation that matches exactly the Mode chosen when you configure the AL Connector.

2. If this name is for example "myCleverOps", then an operation called "myCleverOps" is checked; if found, its schema is returned to the AL Connector.
3. If the name is a derived name (like Iterator, AddOnly and so on) the same is done, even though the actual operations called are the ones listed in Table 1 on page 15.

If neither of the two preceding steps yield a match (for example, because you use an unknown operation) the schema is retrieved form an operation called "querySchema". This operation is never called; it is only used to define a schema that can be retrieved in the AL Connector.

A value of "*" will map **all** attributes.

AssemblyLine Parameters

The target AssemblyLine can be passed a Task Control Block (TCB) as a parameter. This parameter is runtime generated and the AssemblyLine Connector will use this to pass parameters to the target AssemblyLine. This is an alternative to providing parameters through the target AL's "Published AssemblyLine Initialize Parameters" Operation, in conjunction with the **AssemblyLine Parameters** parameter.

See also

Appendix D, “Creating new components using Adapters,” on page 575.

Axis Easy Web Service Server Connector

The Axis Easy Web Service Server Connector is part of the Tivoli Directory Integrator Web Services suite. It is a simplified version of the “Web Service Receiver Server Connector” on page 287 in that it internally instantiates, configures and uses the AxisSoapToJava and AxisJavaToSoap FCs.

Note: Due to limitations of the Axis library used by this component only WSDL (http://www.w3.org/TR/wsdl) version 1.1 documents are supported. Furthermore, the supported message exchange protocol is SOAP 1.1.

The functionality provided is the same as if you chain and configure these FCs in an AssemblyLine which hosts the “Web Service Receiver Server Connector” on page 287. When using this Connector you forgo the possibility of hooking custom processing before parsing the SOAP request and after serializing the SOAP response, that is, you are tied to the processing and binding provided by Axis, but you gain simplicity of setup and use.

The Axis Easy Web Service Server Connector operates in Server mode only.

AssemblyLines support an Operation Entry (op-entry). The op-entry has an attribute *$operation* that contains the name of the current operation executed by the AssemblyLine. In order to process different web service operations easier, the Axis Easy Web Service Server Connector will set the *$operation* attribute of the op-entry.

The Axis Easy Web Service Server Connector supports generation of a WSDL file according to the input and output schema of the AssemblyLine. As in Tivoli Directory Integrator 7.1 AssemblyLines support multiple operations, the WSDL generation can result in a web service definition with multiple operations. There are some rules about naming the operations:

- Pre-6.1 TDI configuration files contain only one input and one output schema referred to as default operation schemas. When a pre-6.1 TDI configuration is used the only operation generated is named as the name of the AssemblyLine as in TDI 6.0.
- In Tivoli Directory Integrator 7.1 configurations if there is an operation named "Default", the corresponding operation in the WSDL file is named as the name of the AssemblyLine.
- In Tivoli Directory Integrator 7.1 configurations if there is an operation named "Default" and there is also an operation with a name as the name of the AssemblyLine, both operations preserve their names in the WSDL file.
- In all other cases the operations appear in the WSDL file as they are named in the AssemblyLine configuration.

This Connector's configuration is relatively simple. The Connector parses the incoming SOAP request, stores it (along with HTTP specific data) into the event Entry and then presents this Entry to the AssemblyLine for Attribute mapping. When the work Entry (now storing the Java representation of the SOAP response) is returned to the Connector in the Response phase, the Connector serializes the response and returns it to the Web Service client.

When this Connector receives a SOAP request, the connector parses it and sets the $operation attribute of the op-entry. The name of the operation is determined by the name of the element nested in the Body element of the SOAP envelope. For parsing the SOAP messages, a SAX parser is used, which compared to a DOM parser adds less performance overhead.

There are several types of SOAP messages:

- When using RPC-style SOAP messages the name of the element is the same as the name of the operation.
- When using Document-style SOAP messages there are two scenarios:

– Using Wrapped Document-style SOAP messages – in this case the body of the SOAP message looks like it is an RPC-style SOAP message; this is achieved by wrapping the contents of the SOAP Body in an element nested in the SOAP Body Element; the name of this element is the name of the SOAP operation.
– Using ordinary, or unwrapped Document-style SOAP messages – in this case the notion of SOAP operation is not defined: the SOAP message is part of some SOAP message exchange. In this case, the Connectors would set the $operation attribute of the op-entry to the name of the element nested in the SOAP Body element and it is the responsibility of you as the Tivoli Directory Integrator developer/deployer to make sure that a Tivoli Directory Integrator solution handles this correctly. When using ordinary or unwrapped Document-style SOAP messages, it is best not to depend on the value of the $operation attribute of the op-entry.

Hosting a WSDL file

The Axis Easy Web Service Server Connector provides the "*wsdlRequested*" Connector Attribute to the AssemblyLine.

If an HTTP request arrives and the requested HTTP resource ends with "?WSDL" then the Connector sets the value of the "*wsdlRequested*" Attribute to **true** and reads the contents of the file specified by the **WSDL File** parameter into the "*soapResponse*" Connector Attribute; otherwise the value of this Attribute is set to **false**.

This Attribute's value thus allows you to distinguish between pure SOAP requests and HTTP requests for the WSDL file. The AssemblyLine can use a Branch Component to execute only the appropriate piece of logic – (1) when a request for the WSDL file has been received, then the AssemblyLine could perform some optional logic or read a different WSDL file and send it back to the web service client, or just rely on default processing; (2) when a SOAP request has been received the AssemblyLine will handle the SOAP request. Alternatively, you could program the `system.skipEntry();` call at an appropriate place (in a script component, in a hook in the first Connector in the AssemblyLine, etc.) to skip further processing and go directly to the Response channel processing.

It is the responsibility of the AssemblyLine to provide the necessary response to a SOAP request.

The Connector implements a public method:

```
public String readFile (String aFileName) throws IOException;
```

This method can be used from Tivoli Directory Integrator JavaScript in a script component to read the contents of a WSDL file on the local file system. The AssemblyLine can then return the contents of the WSDL in the "*soapResponse*" Attribute, and thus to the web service client in case a request for the WSDL was received.

Schema

Input Schema

Table 2. Axis Easy Web Service Server Connector Input Schema

Attribute	Value
host	Type is String. Contains the name of the host to which the request is sent. This parameter is set only if "wsdlRequested" is false.
requestObjArray	The soapRequest represented as an array of objects; converts java.lang.String to Object[] with the perform method of SoapToJava function component.
requestedResource	The requested HTTP resource.
soapAction	The SOAP action HTTP header. This parameter is set only if "wsdlRequested" is false.
soapFault	If a SOAP error occurs, an `org.apache.axis.AxisFault` is stored in this attribute.

Table 2. Axis Easy Web Service Server Connector Input Schema (continued)

Attribute	Value
soapRequest	The SOAP request in txt/XML or DOMElement format. This parameter is set only if "wsdlRequested" is false.
soapResponse	The SOAP response message. If wsdlRequested is true, then soapResponse is set to the contents of the WSDL file.
wsdlRequested	This parameter is true if a WSDL file is requested and false otherwise.
http.username	This attribute is used only when HTTP basic authentication is enabled. The value is the username of the client connected.
http.password	This attribute is used only when HTTP basic authentication is enabled. The value is the password of the client connected.

Output Schema

Table 3. Axis Easy Web Service Server Connector Output Schema

Attribute	Value
responseContentType	The response type.
responseObjArray	The soapRequest represented as an array of objects; the soapResponse gets the value from here, using the JavaToSoap function component to convert Object[] to java.lang.String.
soapFault	If a SOAP error occurs, an `org.apache.axis.AxisFault` is stored in this attribute.
soapResponse	The SOAP response message. If wsdlRequested is true, then soapResponse is set to the contents of the WSDL file.
wsdlRequested	This parameter is true if a WSDL file is requested and false otherwise.
http.credentialsValid	This attribute is used only when HTTP basic authentication is enabled. Its syntax is boolean and if true client authentication is successful. It is responsibility of the AssemblyLine to set this parameter's value when HTTP basic authentication is used.

Configuration

Parameters

TCP Port
The port that the web service will listen for client connections on.

Connection Backlog
This represents the maximum queue length for incoming connection indications (a request to connect). If a connection indication arrives when the queue is full, the connection is refused.

WSDL File
This parameter is required; its type is string. The value of this parameter must be the complete file system path to the WSDL document that describes this web service.

SOAP Operation
The name of the SOAP operation as described in the WSDL file.

Complex Types
This parameter is not required, but if specified it is a list of fully qualified Java class names (including the package name), where the different elements (Java classes) of this list are separated by one or more of the following symbols: a comma, a semicolon, a space, a carriage return or a new line.

Tag Op-Entry
When this parameter is checked (that is, "true") the Connector will tag the op-entry only when

the executed operation is on the list of exposed operations in the AssemblyLine/WSDL. If the operation cannot be found in the WSDL then a SOAP Fault message will be generated and returned to the client.

Note: In TDI 6.0 the AxisEasyWSServerConnector required the **Soap Operation** parameter to be set. In Tivoli Directory Integrator 7.1 when the **Tag Op-Entry** parameter is set to "true" the AxisEasyWSServerConnector will use the extracted operation name instead of the name specified with the **Soap Operation**parameter. In this case the **Soap Operation** parameter is not a required parameter, it can be left blank.

Use SSL
If checked the server will only accept SSL (https) connections. The SSL parameters (keystore, etc.) are specified as values of Java system properties in the `global.properties` file located in the Tivoli Directory Integrator installation folder.

Require Client Authentication
Specifies whether this Connector will require clients to authenticate with client SSL certificates. If the value of this parameter is **true** (that is, checked) and the client does not authenticate with a client SSL certificate, then the Connector will drop the client connection. If the value of this parameter is **true** and the client does authenticate with a client SSL certificate, then the Connector will continue processing the client request. If the value of this parameter is **false**, then the Connector will process the client request regardless of whether the client authenticates with a client SSL certificate.

Auth Realm
This is the basic-realm sent to the client in case authentication is requested. The default is "IBM Tivoli Directory Integrator".

Use HTTP Basic Authentication
This connector supports HTTP basic authentication. To activate, check the **Use HTTP Basic Authentication** checkbox. If activated, the server checks if any credentials are already sent and if not, the server sends authorization request to client. After the client sends the needed credentials, the Connector then sets two attributes: "http.username" and "http.password". These two attributes contain the username and password of the client. It is responsibility of the AssemblyLine to check if this pair of username and password is valid. If the client is authorized successfully then "http.credentialsValid" work Entry Attribute must be set to true. If the client is not authorized then "http.credentialsValid" work Entry Attribute must be set to false. If the client is not authorized then the server sends a "Not Authorized" HTTP message.

Comment
Your own comments go here.

Detailed Log
If checked, will generate additional log messages.

WSDL Output to Filename
The name of the WSDL file to be generated when the "Generate WSDL" button is clicked. This parameter is only used by the WSDL Generation Utility – this parameter is not used during the Connector execution.

Web Service provider URL
The address on which web service clients will send web service requests. Also this parameter is only used by the WSDL Generation Utility – this parameter is not used during the Connector execution.

The **Generate WSDL** button runs the WSDL generation utility.

The WSDL Generation utility takes as input the name of the WSDL file to generate and the URL of the provider of the web service (the web service location). This utility extracts the input and output parameters of the AssemblyLine in which the Connector is embedded and uses that information to

generate the WSDL parts of the input and output WSDL messages. It is mandatory that for each Entry Attribute in the "Initial Work Entry" and "Result Entry" Schema the "Native Syntax" column be filled in with the Java type of the Attribute (for example, "`java.lang.String`"). The WSDL file generated by this utility can then be manually edited.

The operation style of the SOAP Operation defined in the generated WSDL is "*rpc*".

The WSDL generation utility cannot generate a <types...>...</types> section for complex types in the WSDL.

Connector Operation

For an overview of the Axis Easy Web Service Server Connector Attributes, used to exchange information to and from the HTTP/SOAP request, see “Schema” on page 20.

This Connector parses the incoming SOAP request message and stores the Java representation of the SOAP request in the "*requestObjArray*" Connector Attribute. The Connector is capable of parsing both Document-style and RPC-style SOAP messages as well as generating (a) Document-style SOAP response messages, (b) RPC-style SOAP response messages and (c) SOAP Fault response messages. The style of the message generated is determined by the WSDL specified by the **WSDL File** Connector parameter.

The Connector is capable of parsing SOAP request messages and generating SOAP response messages which contain values of complex types which are defined in the <*types*> section of the WSDL document. In order to do that this Connector requires that (1) the **Complex Types** Connector parameter contains the names of all Java classes that implement the complex types used as request and response parameters to the SOAP operation and that (2) these Java classes' class files are located in the Java class path of Tivoli Directory Integrator.

If during parsing the SOAP request an Exception is thrown by the parsing code, then the Connector generates a SOAP Fault Object (`org.apache.axis.AxisFault`) and stores it in the "*soapFault*" Connector Attribute.

This Connector is capable of parsing and generating SOAP response messages encoded using both "literal" encoding and SOAP Section 5 encoding. The encoding of the SOAP response message generated is determined by the WSDL specified by the **WSDL File** Connector parameter.

At the end of AssemblyLine processing in the Response channel phase, this Connector requires the Java representation (*Object[]*) of the SOAP response message from the "*responseObjArray*" Attribute of the *work* Entry to be mapped out. The Connector then serializes the SOAP response message, wraps it into an HTTP response and returns it to the web service client.

See also

“Web Service Receiver Server Connector” on page 287.

Axis2 Web Service Server Connector

The Axis2 Web Service Server Connector can be used to provide a SOAP web service, which is accessible via HTTP/HTTPS.

The logic of such a service is supposed to be implemented as a Tivoli Directory Integrator AssemblyLine, thus leveraging existing Tivoli Directory Integrator components.

The Connector is named after the underlying Axis2 Java library: http://ws.apache.org/axis2/.

Both WSDL 1.1 (http://www.w3.org/TR/wsdl/) and WSDL 2.0 (http://www.w3.org/TR/wsdl20/) documents are supported.

Both SOAP 1.1 and SOAP 1.2 protocols are supported. Only *literal* SOAP messages can be used, *encoded* SOAP messages are not supported. This is a limitation of the underlying Axis2 library (version 1.4.0.1).

The Axis2 Web Service Server Connector supports Server Mode only.

Comparison between Axis1 and Axis2 components

Generally, there are only few cases in which you should use Axis1 components:

- if they need SOAP encoded support
- you may prefer them if you would rather use custom-generated Java types (by the Complex Types Generator FC) instead of attributes with hierarchical structure.

In all other cases the Axis2 components should be used because they:

- support SOAP 1.2 and WSDL 2.0
- provide Schema discovery ("querySchema")
- allows easier manipulation of SOAP headers
- allow control over SOAP faults
- could be enhanced in the future as Axis2 keeps evolving.

SOAP encoding support

The binding in a **WSDL1.1** document describes how the service is bound to a messaging protocol, particularly the SOAP messaging protocol. A WSDL SOAP binding can be either a Remote Procedure Call (RPC) style binding or a document style binding. A SOAP binding can also have an encoded use or a literal use. This gives you four style/use models:

1. RPC/encoded
2. RPC/literal
3. Document/encoded
4. Document/literal

For more information, see http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/.

Support of style/use models in the Tivoli Directory IntegratorAxis2 components is as follows:

1. **RPC/encoded** is not supported due to limitations of the Axis2 library. The RPC-encoded binding is not compliant with WS-I Basic Profile (http://www.ws-i.org/Profiles/BasicProfile-1.1.html#Consistency_of_style_Attribute).
2. **RPC/literal** – supported.
3. **Document/encoded** is not supported but this is not a problem since it is not used at all; in addition, it is not WS-I compliant.
4. **Document/literal** – supported.

In **WSDL 2.0** everything is similar to the document/literal model (all messages are defined directly using a type language, such as XML Schema) so there is no problem with our Axis2 components. As for RPC calls, WSDL2.0 defines a set of rules for designing messages suitable for them. For more information, see http://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/74bae690-0201-0010-71a5-9da49f4a53e2.

Popularity of the RPC/encoded model

All major frameworks for web services support Document/literal messages. Most of the popular frameworks also have some support for rpc/encoded, so developers can still use it to create encoded-only services. As a result it is hard to estimate how many web services, in production use, work only with SOAP encoded messages. However there is a tendency to move away from RPC/encoded towards Document/literal. This is so, because the SOAP encoding specification does not guarantee 100% interoperability and there are vendor deviations in the implementation of RPC/encoded.

Here are some references about encoded support in some popular frameworks:

- Microsoft SOAP Toolkit - the Microsoft SOAP toolkit (used to provide web service access to COM components) by default uses RPC/encoded (supports also Document/literal) as stated here: http://msdn2.microsoft.com/en-us/library/ms995793.aspx.

  The product is deprecated by Microsoft and its support stopped on July 1, 2004: http://msdn2.microsoft.com/en-us/library/aa286526.aspx.
- Microsoft .NET (WSE/WCF) - Since version 1.1 of .NET the default is Document/literal but RPC/encoded is also allowed as stated here: http://msdn2.microsoft.com/en-us/library/dkwy2d72(VS.71).aspx.
- Glassfish (the open-source basis of the Sun Java System Application Server), JBossWS and OracleAS: They all support RPC/encoded to some extent (Document/literal is fully supported): http://wiki.apache.org/ws/StackComparison.

Alternatives:

If you need to use the RPC/encoded model the old web services suite can be used. Also, if you have more information on the service and the SOAP messages, a solution can be created using the HTTP Components and XML Parser.

Using the Connector

The Axis2 Web Service Server Connector is designed for a "**WSDL first**" way of development. This means that the Connector requires a WSDL document describing the web service, so that it knows how clients expect the web service to behave. The implementation of the web service then must stick to the model outlined by the WSDL. (An alternative would be to implement the logic first and have the Connector produce an appropriate WSDL for that implementation.) The reason for this design choice is to make it easy for Tivoli Directory Integrator to fit into an existing communication model by conforming to an already established WSDL description.

For situations where an existing Assembly Line needs to be exposed through a web service interface, Tivoli Directory Integrator offers some basic WSDL generation functionality (see “WSDL Generation” on page 28).

A WSDL document can describe multiple interfaces (or *port types* in WSDL 1.1 terms). Each interface groups a set of operations. One instance of the Axis2 Web Service Server Connector can be used to implement just a single interface. To help the AssemblyLine logic distinguish between different operations, the Connector passes the name of the operation (the local part of the qualified name) in the $operation Attribute of the Operational Entry (op-entry). For more information on AssemblyLine Operations and the Operational Entry see *IBM Tivoli Directory Integrator V7.1 Users Guide*.

Notes:

1. The **SOAP version** depends on the client: If the client sends a SOAP 1.1 request, the Connector will send back a SOAP 1.1 response. If the client sends a SOAP 1.2 request, the Connector will send back a SOAP 1.2 response. The SOAP version settings from the WSDL document are ignored.
2. The Connector does not perform **XML Schema validation** of incoming or outgoing SOAP messages.
3. The Connector does SOAP processing only on **HTTP POST** requests. Other HTTP requests are left for the Assembly Line logic to handle.
4. The Connector will generate a SOAP response only as an answer to a SOAP request: If the HTTP request does not contain a body, the Connector will not generate a SOAP response.
5. The Assembly Line *can override the response* for all requests by specifying an `http.body` Output Attribute. The Connector will not generate a SOAP response if the Assembly Line provides an overriding `http.body` Attribute.
6. For special cases, you can configure the logging level of the underlying Axis2 library in the Log4j configuration file of the Tivoli Directory Integrator Server (`etc/log4j.properties`).

Supported Message Exchange Patterns

The Axis2 Web Service Server Connector supports the following message exchange patterns (described in WSDL 2.0 terms):

In-Only
: The server receives a SOAP request from the client and does not generate any SOAP response; the corresponding WSDL 1.1 term is a "one-way operation".

In-Out
: The server receives a SOAP request and will respond with either a SOAP fault or with a normal SOAP message; the corresponding WSDL 1.1 term is a "request-response operation".

Robust-In-Only
: The server receives a SOAP request from the client and will either respond with a SOAP fault or with no SOAP message at all; there is no corresponding WSDL 1.1 term for this message exchange pattern.

For more information on message exchange patterns see:

http://www.w3.org/TR/wsdl20-adjuncts/#patterns
http://www.w3.org/TR/wsdl#_porttypes

Note: When the server does not generate a SOAP response, it still sends an HTTP response back to the client. In that case the HTTP response body will not contain a SOAP message.

SOAP Faults

You can instruct the Axis2 Web Service Server Connector to generate a SOAP fault in response to a client's request.

This can be achieved using the following Connector attributes:

- $faultCode
- $faultCodeNamespacePrefix
- $faultCodeNamespaceURI
- $faultReason
- $faultNode
- $faultRole
- $faultDetail

See Schema a detailed description of these and other attributes.

For more information on SOAP faults see:

http://www.w3.org/TR/soap12-part1/#soapfault
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383507

SOAP Headers

The Connector provides access to the SOAP header of the SOAP request for analysis, in case of special or advanced use.

It also allows user-defined SOAP headers to be included in the response.

Note that any user-defined SOAP headers affect both normal SOAP messages and SOAP faults.

See section "Schema" for a detailed description of the attributes.

The HTTP Transport Layer

The Axis2 Web Service Server Connector uses the "HTTP Server Connector" on page 115 as its HTTP transport.

In special, advanced cases you can take advantage of the control that the HTTP Server Connector provides over the HTTP request and the HTTP response.

You can analyze the HTTP headers of the request and set the HTTP headers of the response.

You can even override the whole HTTP body of a response. The Axis2 Web Service Server Connector parses SOAP messages out of HTTP POST requests only. HTTP GET requests are not processed by the SOAP engine, and you are free to implement your own logic in such cases – for example you can return a WSDL document if an HTTP GET request arrives with an URI that ends with "`?wsdl`".

WSDL Generation

The Axis2 WS Server Connector requires a WSDL document in order to function. If you have a working AssemblyLine but you do not have a WSDL document, you can use Tivoli Directory Integrator to generate one, using an instance of this Connector. The service name in the generated WSDL document will be set to the name of the AssemblyLine.

Note that the WSDL generation functionality is aimed at novice users as a quick start. If you have some web service expertise, we strongly recommend that you design the WSDL document yourself or at least thoroughly inspect the generated WSDL document before putting it into production use.

To generate a WSDL file:

1. Add an instance of the Axis2 WS Server Connector to the AssemblyLine for which you want to generate a WSDL document.
2. Fill in the **WSDL Output to Filename**, **Web Service provider URL** and **WSDL Version** parameters.
3. Press the **Generate WSDL** button.

Schema

Input Schema

See the documentation of the "HTTP Server Connector" on page 115 for transport related attributes.

You can add attributes such as `http.content-type` and `http.content-length` to the Input Map and use these parameters of the SOAP request in the logic of the AssemblyLine.

Another useful attribute is `http.method`, which holds the type of request received by the server (GET or POST). Since the connector parses only POST requests, the value of this attribute can be checked and in case of a GET request a specific return value set (for example the WSDL document describing the service or an HTML document).

`http.SOAPAction` is also a significant HTTP header that is important for the Web Services. It is set in the HTTP binding of the SOAP message, and its value is an URI. Some SOAP bindings do not require a SOAPAction and omit this attribute.

A SOAP Message Embedded in an HTTP Request:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

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

See the Schema section of the Axis2 WS Client Function Component for a description of WSDL-specific Attributes, such as the incoming message.

$soapHeader

A Hierarchical Attribute, which contains the SOAP header of the incoming SOAP message.

For example, consider the following SOAP message:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

 <soapenv:Header
  xmlns:myns="http://www.my.com">
  <myns:myheader />
 </soapenv:Header>

 <soapenv:Body><payload /></soapenv:Body>

</soapenv:Envelope>
```

Then the **$soapHeader** Hierarchical Attribute will be a DOM representation of the following XML element:

```
<$soapHeader
 xmlns:myns="http://www.my.com">
 <myns:myheader />
</$soapHeader>
```

Note that the only difference between the **$soapHeader** element and the Header element from the SOAP message is that the **$soapHeader** element does not have an associated namespace. The idea is to save you from considering what namespace to use (the namespace of the header element differs between SOAP versions).

Output Schema

See the documentation of the "HTTP Server Connector" on page 115 for transport related attributes.

The HTTP attributes can be used not only to set characteristics of the SOAP response, but to alter the behavior of the AssemblyLine. For instance when the attribute `http.body` is mapped in the Output Map of the connector, its value is directly set as SOAP response and the Axis2 engine is not used to generate it. A similar technique is used in the first of the shipped examples. There the value of `http.method` is

checked and in case of a GET request the `http.body` attribute is set with the contents of the WSDL file describing the service. If a POST request is received a SOAP response is assembled and sent.

Another useful HTTP attribute is `http.status`. It can be mapped to the Output Map of the connector and its value set according to the AssemblyLine logic. This way you can modify the status of the HTTP response that the server will send. Set "200" for OK, "403" for Forbidden, "404" for Not Found, and so forth.

See the Axis2 WS Client Function Component Schema section for description of WSDL-specific Attributes, such as the incoming message.

$authResult

This optional Attribute represents the result of the authentication of the current client.

If the Attribute is set to "true" (case insensitive), the authentication of the client is considered successful, otherwise (if the Attribute is missing or has some other value) the Connector assumes the authentication has failed and returns an error HTTP response to the client.

For more information on authentication see section “Authentication” on page 34.

$soapHeader

A Hierarchical Attribute, whose child elements will be added to the SOAP header of the outgoing SOAP message.

For example consider the following SOAP message:

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

 <soapenv:Header
  xmlns:myns="http://www.my.com">
  <myns:myheader />
 </soapenv:Header>

 <soapenv:Body><payload/></soapenv:Body>

</soapenv:Envelope>
```

Suppose that the **$soapHeader** Hierarchical Attribute is the DOM representation of the following XML element:

```
<$soapHeader
 xmlns:otherns="http://www.other.com">
 <otherns:otherheader>
  This is an example of a user-defined SOAP header.
 </otherns:otherheader>
</$soapHeader>
```

When combining the above SOAP message with the contents of the **$soapHeader** Hierarchical Attribute, the result will be the following SOAP message:

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

 <soapenv:Header
  xmlns:myns="http://www.my.com"
  xmlns:otherns="http://www.other.com">
  <myns:myheader />
  <otherns:otherheader>
   This is an example of a user-defined SOAP header.
  </otherns:otherheader>
 </soapenv:Header>

 <soapenv:Body><payload/></soapenv:Body>

</soapenv:Envelope>
```

$faultCode

This is a mandatory Attribute, if you want the Connector to generate a SOAP fault response.

The **$faultCode** Attribute is designed to be used in combination with the **$faultCodeNamespacePrefix** and **$faultCodeNamespaceURI**. Together these three Attributes can fully define a qualified name – that is a namespace URI ($faultCodeNamespaceURI), a local part ($faultCode) and a namespace prefix ($faultCodeNamespacePrefix). This qualified name represents the code of the SOAP fault.

When working with SOAP 1.2, the **$faultCode** Attribute will be used as the local part of the qualified name, which appears inside the **Value** element, which is a child of the **Code** element, which is a child of the **Fault** element.

For example, if the **$faultCode** Attribute contains the string "mycode" and the **$faultCodeNamespacePrefix** and **$faultCodeNamespaceURI** Attributes are missing, the fault element inside the body of the SOAP message would look like this:

```
<soapenv:Fault xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
 <soapenv:Code>
  <soapenv:Value>mycode</soapenv:Value>
 </soapenv:Code>
 ...
</soapenv:Fault>
```

On the other hand, if you set all three Attribute like this:

```
$faultCodeNamespaceURI = http://www.w3.org/2003/05/soap-envelope
$faultCode = Sender
$faultCodeNamespacePrefix= env
```

then this is what the SOAP fault will look like:

```
<soapenv:Fault xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
  xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <soapenv:Code>
  <soapenv:Value>env:Sender</soapenv:Value>
 </soapenv:Code>
 ...
</soapenv:Fault>
```

When working with SOAP 1.1, the **$faultCode** Attribute will be used as the content of the $faultcode element, which is a child of the **Fault** element.

For example if you set the following Attribute values:

```
$faultCodeNamespaceURI = http://www.my.com
$faultCode = myfaultcode
$faultCodeNamespacePrefix = mypref
```

then the fault element inside the SOAP body will look like this:

```
<soapenv:Fault xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:mypref="http://www.my.com">
 <faultcode>mypref:myfaultcode</faultcode>
 ...
</soapenv:Fault>
```

In general, you might prefer to use some of the predefined SOAP fault codes rather than make up your own:

http://www.w3.org/TR/soap12-part1/#soapfault (for SOAP 1.2)
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383507 (for SOAP 1.1).

$faultCodeNamespaceURI

This optional Attribute represents the URI of the namespace of the SOAP fault code. It is used in

combination with the **$faultCode** and **$faultCodeNamespacePrefix** Attributes. For more information see the description of the **$faultCode** Attribute.

$faultCodeNamespacePrefix
This optional Attribute represents the namespace prefix of the SOAP fault code. It is used in combination with the **$faultCode** and **$faultCodeNamespaceURI** Attributes. For more information see the description of the **$faultCode** Attribute.

$faultReason
This is a mandatory Attribute, if you want the Connector to generate a SOAP fault response.

The **$faultReason** Attribute should contain a human-friendly description of the SOAP fault. When working with SOAP 1.2, the value of the **$faultReason** Attribute is used as the content of the first Text element, which is a child of the Reason element inside the Fault element.

For example, if you set the **$faultReason** Attribute to "myreason", the SOAP fault element will look like this:

```
<soapenv:Fault xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
  ...
  <soapenv:Reason>
   <soapenv:Text
    xml:lang="en-US"
    xmlns:xml="http://www.w3.org/XML/1998/namespace">
    myreason
   </soapenv:Text>
  </soapenv:Reason>
  ...
</soapenv:Fault>
```

Note that the language will always be set to "en-US". When working with SOAP 1.1, the value of the **$faultReason** Attribute is used as the content of the **faultstring** element inside the **Fault** element.

For example, if you set the **$faultReason** Attribute to "myreason", the SOAP fault element will look like this:

```
<soapenv:Fault xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  ...
  <faultstring>myreason</faultstring>
  ...
</soapenv:Fault>
```

$faultNode
This optional Attribute is an URI, which represents the SOAP node that generated the fault.

The **$faultNode** Attribute corresponds to the **Node** element in SOAP 1.2 and the **faultactor** element in SOAP 1.1.

$faultRole
This optional Attribute is an URI, which represents the role the node was operating in at the point the fault occurred.

The **$faultRole** Attribute corresponds to the **Role** element in SOAP 1.2.

For more information on SOAP roles see http://www.w3.org/TR/soap12-part1/#soaproles.

When working with SOAP 1.1 the value of this Attribute is ignored.

$faultDetail
This optional Hierarchical Attribute represents additional application-specific information describing the fault.

The first child element of the **$faultDetail** Hierarchical Attribute will be used as the content of the **Detail** element in SOAP 1.2 or the **detail** element in SOAP 1.1.

For example if the **$faultDetail** Hierarchical Attribute is the DOM representation of the following XML element:

```
<$faultDetail>
  <myfaultdata>some information here</myfaultdata>
</$faultDetail>
```

then the SOAP fault element will look like this (using SOAP 1.2):

```
<soapenv:Fault xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
  ...
  <soapenv:Detail>
   <myfaultdata>some information here</myfaultdata>
  </soapenv:Detail>
  ...
</soapenv:Fault>
```

Using SOAP 1.1, it will look like this:

```
<soapenv:Fault xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  ...
  <detail>
   <myfaultdata>some information here</myfaultdata>
  </detail>
  ...
</soapenv:Fault>
```

Configuration

The Axis2 Web Service Server Connector has the following parameters:

WSDL URL
The location of the WSDL document, which contains a description of the web service. The Connector will ignore the endpoint address information set in the WSDL document. Features such as listening port, SSL and HTTP basic authentication can be controlled only by means of the Connector parameters.

Service
The name of the service description in the WSDL document. This parameter has an associated script, which lists all available service descriptions in the WSDL document. The script requires the **WSDL URL** parameter to be set first.

If the WSDL document contains a description of a single service, this parameter can be left empty. On the other hand, if the WSDL document describes multiple services, this parameter is mandatory.

TCP Port
The TCP port to listen for incoming requests (the default port is **80**).

Connection Backlog
This represents the maximum queue length for incoming connection indications (a request to connect). If a connection indication arrives when the queue is full, the connection is refused.

HTTP Basic Authentication
If enabled (by default it is not), clients will be challenged for HTTP Basic authentication.

Auth Realm
The authentication realm sent to the client when requesting HTTP Basic authentication. The default is "IBM Tivoli Directory Integrator".

Use SSL
If enabled (by default it is not), then the Connector will require clients to use SSL; non-SSL connection requests will fail.

When SSL is used, the Connector will use the default Tivoli Directory Integrator Server SSL settings – certificates, keystore and truststore.

Require Client Authentication
If enabled (by default it is not), the Connector mandates client authentication when using SSL. This means that the Connector will require clients to supply client-side SSL certificates that can be matched to the configured Tivoli Directory Integrator trust store. This parameter is only taken into account if the previous parameter (Use SSL) is enabled as well.

WSDL File Generator Parameters:

The following parameters are related to WSDL file generation (they are not used at runtime):

WSDL Output to Filename
The name of the WSDL file to be generated. This parameter specifies the name of the WSDL file that is generated when the WSDL file generation utility is run.

Web Service provider URL
The address to which web service clients will send web service requests. This value is used only by the WSDL Generation Utility.

WSDL Version
The version of the WSDL document that is generated. You can select from the following values:

- **1.1** – for WSDL 1.1
- **2.0** – for WSDL 2.0

Generate WSDL
The button that causes the WSDL Generation Utility to be run. The output is sent to the file specified in the **WSDL Output to Filename** parameter.

Security and Authentication

Encryption

The Axis2 Web Service Server Connector supports transport level security by the use of SSL/TLS.

To turn SSL on, set the **Use SSL** Connector parameter to true.

To turn SSL client authentication on, set the **Require Client Authentication** Connector parameter to true.

For more information on Connector parameters, see "Configuration" on page 33.

Authentication

By default the Axis2 Web Service Server Connector has HTTP basic authentication disabled.

To turn HTTP basic authentication on, set the **HTTP Basic Authentication** Connector parameter to true. Also set the **Auth Realm** to the name of the authentication realm – the client will be prompted to authenticate against that realm.

For more information on Connector parameters see "Configuration" on page 33.

The following Connector Attributes are related to HTTP basic authentication:

Input Schema:

http.remote_user
This is the username specified in the HTTP client request.

http.remote_pass
This is the password specified in the HTTP client request.

Output Schema:

$authResult

This is the result of the authentication process. The AssemblyLine associated with the Connector should set this Attribute according to the result of any user-defined authentication logic.

Note that the actual authentication logic must be implemented in the associated Assembly Line, for example by verifying client credentials against a database or an LDAP server.

Authorization

You can implement custom authorization logic in the Connector's associated AssemblyLine, based on the username (see “Authentication” on page 34) provided by the Connector.

See also

The example in *TDI_install_dir*/examples/axis2_web_services.

Command line Connector

The command line Connector enables you to read the output from a command line or pipe data to a command line's standard input. Every command argument is separated by a space character, and quotes are ignored. The command is executed on the local machine.

Note: You do not get a separate shell, so redirection characters (| > and so forth) do not work. To use redirection, make a shell-script (UNIX) or batch command (DOS) with a suitable set of parameters. For example, on a Windows system, type

```
cmd /c dir
```

to list the contents of a directory.

The Connector supports Iterator and AddOnly mode, as well as CallReply mode.

In Iterator and AddOnly mode, the command specified by the **Command Line** parameter is issued to the target system during Connector initialization, which implies it will only be issued once for the whole AssemblyLine lifetime.

However, in CallReply mode, the command is issued to the target system on each iteration of the AssemblyLine, after Output Attribute Mapping (call phase), and before Input Attribute Mapping (reply phase). In this mode, you must provide the command to be executed in an attribute called **command.line**; after it has executed you will find the output result in an attribute called **command.output**.

If a Parser is attached to the Command Line Connector, the output result will be parsed.

Native-encoded output on some operating systems

When you use the Command Line Connector to run a program on a Windows operating system, the output from the program might be encoded using a DOS code page. This can cause unexpected results, because Windows programs usually use a Windows code page. Because a DOS code page is different from a Windows code page, it might be necessary to set the Character Encoding in the Command Line Connector's Parser to the correct DOS code page for your region; for example: `cp850`.

The same issue may arise on for example i5/OS®; here the output from commands is usually encoded in the IBM037 character set; and on z/OS® it could be EBCDIC.

Also see “Character Encoding conversion” on page 295.

Some words on quoting

On Linux/Unix systems, this Connector has the capability to attempt to deal with the quoting of parameters that may contain lexically important characters. When the parameter **Use sh** is checked, Tivoli Directory Integrator uses the `sh` program (for example, the standard Linux shell) to run the command line, and sh will handle quoting as you expect. If you do not have sh on your operating system, do not check this box.

Without using sh, when the Command Line Connector is run on a Unix/Linux platform, it does not handle a command line with a parameter in quotes correctly. For example, the command:

```
Report -view fileView -raw -where "releaseName = 'ibmdi_60' and nuPathName like 'src/com/ibm/di%' "
```

This command should have the phrase `"releaseName = 'ibmdi_60' and nuPathName like 'src/com/ibm/di%' "` as one parameter, but it does not. The reason is that Tivoli Directory Integrator uses the Java Runtime exec() method, which splits all commands at spaces, and ignores all quoting. We would have liked this to be split according to the quotes. Checking **Use sh** (when possible) solves this problem.

Configuration

The Connector needs the following parameters:

Command Line
The command line to run. Used for Iterator and AddOnly modes only.

Use sh
If enabled (by default it is not), will instruct the Connector to use sh-like parsing. Specifically, when this parameter is set to true the Connector is able to correctly parse quoted (using double-quotation marks) command line arguments which contain spaces.

This feature is only available on operating systems which provide the "sh" shell command interpreter (usually UNIX-like operating systems).

Detailed Log
If this field is checked, additional log messages are generated.

Parser The Parser responsible for interpreting or generating entries.

Examples

Refer to the *root_directory*/examples/commandLine_connector directory of your IBM Tivoli Directory Integrator installation.

See also

"Remote Command Line Function Component" on page 431,
"z/OS TSO/E Command Line Function Component" on page 437

Database Connector

The Database connector is a simplified version of the “JDBC Connector” on page 133. The Database connector is intended to provide a more user-friendly user interface to configuring a database. All advanced parameters are removed from the connector configuration form and the most commonly used parameters are the only available ones.

For behaviour, Connector Modes and so forth, see the appropriate sections of the “JDBC Connector” on page 133. Specifically, for information on where to place driver libraries in order to be able to set up a connection to a database system, see “Understanding JDBC Drivers” on page 133; however, the Database connector automates some of the process of creating a JDBC URL.

Configuration

The Database Connector requires the following parameters:

Database Type
: This section prompts you for hostname, port and database parameters. Based on the database type dropdown selection the `jdbcDriver` and `jdbcSource` parameters are generated.

Username
: Signon to the database using this username; only the tables accessible to this user will be shown. This reflects the `jdbcuser` parameter of the JDBC connector.

Password
: The password used in the signon for the user.

Table Name
: The name of the table you wish to access with this connector. You can use the **Select** button to query the database for accessible tables, provided you are able to signon to the database.

Direct TCP /URL scripting

You might want to access URL objects or TCP ports directly, not using the Connectors. The following code is an example that can be put in your Prolog:

TCP

```
// This example creates a TCP connection to www.example_page_only.com
  and asks for a bad page

var tcp = new java.net.Socket ( "www.example_page_only.com", 80 );
var inp = new java.io.BufferedReader ( new java.io.InputStreamReader
  ( tcp.getInputStream() ) );
var out = new java.io.BufferedWriter ( new java.io.OutputStreamWriter
  ( tcp.getOutputStream() ) );

task.logmsg ("Connected to server");

// Ask for a bad page
out.write ("GET /smucky\r\n");
out.write ("\r\n");

// When using buffered writers always call flush to make sure data
  is sent on connection
out.flush ();

task.logmsg ("Wait for response");
var response = inp.readLine ();

task.logmsg ( "Server said: " + response );
```

URL

```
// This example uses the java.net.URL object instead of the raw
  TCP socket object

var url = new java.net.URL("http://www.example_page_only.com");
var obj = url.getContent();

var inp = new java.io.BufferedReader ( new java.io.InputStreamReader
  ( obj ) );
while ( ( str = inp.readLine() ) != null ) {
task.logmsg ( str );
}
```

Domino/Lotus Notes Connectors

In order to connect to a Domino Server or a Lotus Notes system, a discussion on what types of connections ("Session types" in Lotus® Notes® terminology) are possible, is appropriate. For these Connectors to operate, you will need to install a Domino/Lotus Notes client library, and the decision on which client library to install (also see “Post Install Configuration” on page 45) hinges on which Session Type is required.

Session types

Local Client Session

Local client session calls to the Domino Server are based on the Notes user ID.

A Notes client must be installed locally. This session type requires the `Notes.jar` file to be present in the *`TDI_install_dir`*`/jars/3rdparty/IBM` folder and that the local client binaries are specified in the PATH system environment variable.

Local Server Session (Domino Local Session)

When creating this type of session, the ID file of the local server is used.

The host parameter in the Notes API method for creating session must be null. A reference to the current server such as a null server parameter in the session creation method means the local Domino environment is indicated. If a Local Client session is to be created, the user parameter is also required to be null, which indicates to use the Notes user ID.

The local server is used only to create a session. However, servers connected to the local environment can still be accessed by specifying their names. The name is pointed as first parameter of the *getDatabase* methods of the `lotus.domino.Session` class.

For Local Server sessions you need to install Lotus Domino Server on the machine where Tivoli Directory Integrator is installed. Also, the path to this server installation must be added to the PATH system environment variable.

IIOP Session and the IOR Parameter

An IIOP Session is a network based session, where the remote Domino server handles the client requests.

When an IIOP session is specified the Connector uses a Domino User Name and the Internet password of this user for authentication. The users' User Name and Internet password are parameters of the Connector. It is not necessarily the same user as the system local user ID.

The IOR is a text string required by the Domino Java API in order to establish an IIOP session to the Domino Server. A client, like a Lotus Notes Connector, decodes the string IOR and uses it to establish the remote session. It is contained in a file, called `diiop_ior.txt`.

Any of the following changes could make the IOR string stale:

- Changing a DIIOP port number
- Enabling or disabling a DIIOP port
- Changing the TCP/IP address

There are two approaches for the creation of an IIOP Session:

Provide the IOR String explicitly

The TDI 6.0 Domino Change Detection Connector uses a session creation method which obtains the IOR string from the Domino HTTP task. In Tivoli Directory Integrator 7.1 Domino/Lotus Connectors the parameter **IOR String** is externalized. This parameter is optional. If this parameter is missing or has no value, IIOP sessions will be created as they used to in TDI 6.0. If this parameter is present in the Connector configuration the following methods from the Domino Java API will be used for session creation:

```
static public Session createSessionWithIOR(String IOR,
        String user, String passwd)
        throws NotesException

    static public Session createSessionWithIOR(String IOR,
        String args[], String user, String passwd)
        throws NotesException
```

Providing this Connector parameter improves the Connectors in two ways:

- It is no longer required that the Domino HTTP task be running in order for the Connector to function, thus lowering the Connector setup requirements.
- The Connector will be able to function when the Domino HTTP task is configured to use the SSL port only.

Get the IOR String from the HTTP task

In this case, the **HTTP Port** parameter is used by the Connector to get the IOR String from the Domino Server using its HTTP task. If the Connector is to use the local client so as to create a session to the Domino Server, this port is not taken into account.

When creating an IIOP session SSL could be used. The Connector first tries to retrieve the IOR String from the HTTP Task and then use the session creation method that accepts an array of strings as a parameter by providing the retrieved IOR instead of host parameter.

If SSL is used, the Connector tries to create a session using the following method:

```
static public Session createSessionWithIOR(String ior,
     String user, String passwd)
      throws NotesException
```

If SSL is not used, the Connector tries to create a session using the following method:

```
static public Session createSession(String host, String args[],
     String user, String password)
     throws NotesException
```

In this case the value of the **HTTP Port** parameter is appended to the host. This method tries to get the IOR string from the Domino HTTP task that should run on this port. The task must not use this port to run SSL on it.

These session types require `ncso.jar` file to be present in the *TDI_install_dir*`/jars/3rdparty/IBM` folder and that the local server binaries are specified in the PATH system environment variable.

Supported session types by Connector

Table 4. Supported Domino/Lotus Notes Session types, per Connector

Supported Sessions ► Connectors ▼	Local Client Session	Local Server Session	IIOP session
Domino Change Detection Connector	Yes	No	Yes
Domino Users Connector	Yes	Yes	Yes
Lotus Notes Connector	Yes	Yes	Yes
Domino AdminP Connector	No	Yes	Yes

Note: The Domino APIs for SSL are not JSSE compliant, and are instead Domino specific. This means that the Tivoli Directory Integrator truststore and keystore do not play any part in SSL configuration for the Domino Change Detection Connector. For SSL configuration of the Domino Change Detection Connector, the `TrustedCerts.class` file that is generated every time the DIIOP process starts (in the Domino Server) must be in the classpath of

Tivoli Directory Integrator (ibmditk or ibmdisrv). You must copy the `TrustedCerts.class` to a local path included in the CLASSPATH or have the `Lotus\Domino\Data\Domino\Java` of your Domino installation in the CLASSPATH.

The file must be loaded by the same class loader that loads the `ncso.jar` file. As the `ncso.jar` file is loaded by the system class loader in Tivoli Directory Integrator, the `TrustedCerts.class` file must be loaded by the system class loader. This can be easily done by dropping the `TrustedCerts.class` file in the "classes" folder; see “"Classes" folder” on page 48 for more information.

Local session on a 64-bit operating system

In general, IBM Tivoli Directory Integrator with a 32/64-bit JVM can establish a Local Session (client or server) only using a corresponding 32/64-bit Domino®/Notes installation:

Table 5.

	32-bit Domino/Notes	64-bit Domino*
32-bit JVM	Yes	No
64-bit JVM	No	Yes

(*) Currently (version 8.5) Lotus does not provide 64-bit Notes clients.

Local Client and Local Server sessions use Lotus' Java API (Notes.jar). This Java API relies on the native libraries of the Notes/Domino installation (that is why you must have either a Notes Client or a Domino Server installed on the same machine as IBM Tivoli Directory Integrator). Most modern operating systems allow a process to use either 32-bit or 64-bit native binaries but not both. If you use a 32-bit executable to start a process, that process can use only 32-bit native libraries.

On most 64-bit platforms it is possible to install and use 32 bit applications. If you want to use Local Server (Local Client) session to a 32-bit Domino Server (Notes Client) (even on a 64-bit operating system), you need to use IBM Tivoli Directory Integrator with a 32-bit JVM.

Note: The Domino/Notes Connectors are supported on Domino R8 and Domino R8.5 only.

Post Install Configuration

There are a few configuration steps which must be performed after Tivoli Directory Integrator has been installed so that the Lotus Notes/Domino Connectors can run.

Verify that the version of the Domino Server or Lotus Notes client (the Connector will be used with) is supported.

When a Connector is deployed on a Notes client machine these steps need to be performed only on the Notes client machine and not on the Domino Server machine to which the Notes client is connected.

When a Connector is deployed on a Domino Server machine these steps need to be performed on that Domino Server machine.

Lotus provides a Java library called `Notes.jar`, which provides interfaces to native calls that access the Domino Server (possibly through the network). It can be found in the folder where the Domino Server or the Lotus Notes client is installed (for example, `C:\Lotus\Domino` or `C:\Lotus\Notes`). Different settings must be performed depending on whether you create Local Client Session or Local Server Session, because the binaries differ between Lotus Domino and Lotus Notes.

If you create Local Client Session, perform the following:

- Copy `Notes.jar` to *`TDI_install_dir`*`/jars/3rdparty/IBM` or to the folder specified by the `com.ibm.di.loader.userjars` property in `global.properties` (or `solution.properties`).

- Ensure the ncso.jar file does not exist in the *TDI_install_dir*/jars folder and any of its subfolders.
- Add the path to the local Notes binaries (for example, C:\Lotus\Domino or C:\Lotus\Notes) to the PATH environment variable inside the ibmditk (or ibmditk.bat) and ibmdisrv (or ibmdisrv.bat) shell scripts. Then add the following parameter to the list of parameters supplied to the java executable at the bottom of the two shell scripts:

  ```
  -Djava.library.path="%PATH%"
  ```

- If you are getting the following exception:

  ```
  CTGDJE010E Connector was unable to initialize local Notes session to Domino Server.
  Exception encountered: java.lang.Exception: Native call SECKFMSwitchToIDFile failed with error: code 259, 'File does not exist'
  ```

  You need to copy the Notes ID file intended for authentication to the Domino server in the Solution directory of Tivoli Directory Integrator. Usually this file is located in the folder *Notes_install_dir*/Data/. The problem is caused by the way the notes.ini file points to the ID file. Instead of an absolute path a relative one is used (KeyFilename=user.id), so our components look for it in their current directory - the solution directory of Tivoli Directory Integrator and not in the Data folder of the Notes installation.

If you create Local Server Session, perform the following:

- Ensure the ncso.jar file does not exist in the *TDI_install_dir*/jars folder and any of its subfolders.
- Copy Notes.jar to *TDI_install_dir*/jars/3rdparty/IBM or to the folder specified by the com.ibm.di.loader.userjars property in global.properties (or solution.properties).
- Add the path to the local Domino binaries (for example, C:\Lotus\Domino) to the PATH environment variable inside the ibmditk (or ibmditk.bat) and ibmdisrv (or ibmdisrv.bat) shell scripts. Then add the following parameter to the list of parameters supplied to the java executable at the bottom of the two shell scripts:

  ```
  -Djava.library.path="%PATH%"
  ```

Note: Because of the way the Notes API is implemented, there can only exist one of two jars: either ncso.jar or Notes.jar in *TDI_install_dir*/jars/3rdparty/IBM. If both jars are present, then the behavior of the Connector is unpredictable.

If you create IIOP Session, perform the following:

- Ensure the Notes.jar file does not exist in the *TDI_install_dir*/jars folder and any of its subfolders.
- Copy ncso.jar to *TDI_install_dir*/jars/3rdparty/IBM or to the folder specified by the com.ibm.di.loader.userjars property in global.properties (or solution.properties).
- Due to limitations of the native library you will need to add the local Domino binaries (for example, C:\Lotus\Domino) to the PATH environment variable inside the ibmditk (or ibmditk.bat) and ibmdisrv (or ibmdisrv.bat) shell scripts.
- If you are getting the following exception:

  ```
  CTGDJE010E Connector was unable to initialize local Notes session to Domino Server.
  Exception encountered: java.lang.Exception: Native call SECKFMSwitchToIDFile failed with error: code 259, 'File does not exist'
  ```

  You need to copy the Notes ID file intended for authentication to the Domino server in the Solution directory of Tivoli Directory Integrator. Usually this file is located in the folder *Notes_install_dir*/Data/. The problem is caused by the way the notes.ini file points to the ID file. Instead of an absolute path a relative one is used (KeyFilename=user.id), so our components look for it in their current directory – TDI's solution directory and not in the Data folder of the Notes installation.

Native API call threading

When an AssemblyLine (containing Connectors) is executed by the Tivoli Directory Integrator Server it runs in a single thread and it is only the AssemblyLine thread that accesses the AssemblyLine Connectors. The initialization of the Notes API, selecting entries, iterating through the entries and the termination of the Connector is performed by one worker thread.

A requirement of the Notes API is that when a **local session** is used each thread that executes Notes API functions must initialize the NotesThread object, before calling any Notes API functions. The Config Editor GUI threads do not initialize the NotesThread object and this causes a Notes exception.

There are several ways to initialize the NotesThread object. The way the Connectors do it is to call the `NotesThread.sinitThread()` method when a local session is created.

That is why the all Lotus Notes and Domino Connectors use their own internal thread to initialize the Notes runtime and to call all the Notes API functions. The internal thread is created and started on Connector initialization and is stopped when the Connector is terminated. The Connector delegates the execution of all native Notes API calls to this internal thread. The internal thread itself waits for and executes requests for native Notes API calls sent by other threads.

This implementation makes Connectors support the Config Editor GUI functionality and multithread access in general. The Lotus Notes Connector initializes the Notes runtime if a local session is created.

The ncso.jar file

In order to use IIOP sessions, the Tivoli Directory Integrator Lotus Notes/Domino components require the presence of the `ncso.jar` file.

From IBM Tivoli Directory Integrator 6.1, `ncso.jar` will no longer be shipped with the Tivoli Directory Integrator product. You need to manually provide this file in order for the Tivoli Directory Integrator Lotus/Domino components to function properly.

However, the `ncso.jar` file is shipped with the Domino Server. This file can be taken from the Domino installation (usually `<Domino_root>\Data\domino\java\ncso.jar` on Windows platforms) and place it in the Tivoli Directory Integrator_root\jars\3rdparty\IBM folder, so that the Tivoli Directory Integrator Server will load it on initialization. Since the `ncso.jar` will not be provided as part of the IBM Tivoli Directory Integrator 7.1 installation, some existing Tivoli Directory Integrator 6.0 functionality will change as follows.

Server aspects

The "-v" command-line option: The Tivoli Directory Integrator Server provides the "-v" command-line option which displays the versions of all Tivoli Directory Integrator components. Since the `ncso.jar` file will not be provided as part of the Tivoli Directory Integrator installation, if `ncso.jar` is not taken from the Domino server or Lotus Notes installation, messages like the following will be displayed (The components which do not rely on the `ncso.jar` have their versions displayed properly):

```
ibmdi.DominoUsersConnector:
com.ibm.di.connector.dominoUsers.DominoUsersConnector:
2006-03-03: CTGDIS001E The version number of the Connector is undefined
```

The Server API getServerInfo method: The Server API provides a method to request version information about Tivoli Directory Integrator components (Session.getServerInfo). If version information is requested via the Server API about any of the Connectors which rely on `ncso.jar` and if this jar is not taken from the Domino server or Lotus Notes installation, an error is thrown. For example if the local Server API is accessed through a script like this:

```
session.getServerInfo().getInstalledConnectors()
```

the following error is displayed:

```
18:16:12  CTGDKD258E Could not retrieve version info for class
'com.ibm.di.connector.DominoChangeDetectionConnector'.:
java.lang.NoClassDefFoundError: lotus.domino.NotesException
```

Running an AssemblyLine, IIOP Session

AssemblyLines which use a Connector (which uses an IIOP session) will fail to execute with a NoClassDefFoundError exception, if the `ncso.jar` file is not taken from the Domino Server or Lotus Notes installation.

Reported component availability

Component version table: This is the table with the versions of all installed Tivoli Directory Integrator components (available from the context-menu option **Show Installed Components** on a server visible in the **Servers** panel.) This table will fail to display component versions for any of the Notes/Domino Connectors if neither the `Notes.jar` nor the `ncso.jar` is taken from the Domino/Notes installation

Component combo box: The Insert new object box (activated by choosing **Insert Component...**, where available) will display all existing Tivoli Directory Integrator Connector modes (not only the supported ones) for the Notes/Domino Connectors, if neither the `Notes.jar` nor the `ncso.jar` is taken from the Domino/Notes installation.

"Input Map" connection to the data source: Attempting a connection to the data source from the Input Map tab for any of the Notes/Domino Connectors will display an error that the Connector could not be loaded, if the jar library is not taken from the Notes/Domino installation, whatever session is created.

"Classes" folder

This folder, located at *TDI_Install_Folder*`/classes` contains user-provided class files that are loaded by the system class loader. This folder can be used for specifying custom classes which must be loaded by the system class loader.

In order for a class file to be loaded by the system class loader, the class file needs to be copied to this folder. If the class is inside a Java package, then the class file must be put in the corresponding folder under the "classes" folder. For example, if a class file is contained in a Java package named `com.ibm.di.classes`, then the class file must be put inside the *TDI_Install_Folder*`/classes/com/ibm/di/classes` folder.

Only class files in the "classes" folder are loaded. That means that if a jar file is located in this folder, it will not be loaded at all. If the classes are packaged in a jar file, then these classes need to be extracted from the jar file into the "classes" folder.

Domino Change Detection Connector

The Domino Change Detection Connector enables IBM Tivoli Directory Integrator 7.1 to detect when changes have occurred to a database maintained on a Lotus Domino server. The Domino Change Detection Connector retrieves changes that occur in a database (NSF file) on a Domino Server. It reports changed Domino documents so that other repositories can be synchronized with Lotus Domino.

Notes:

1. Due to the Lotus Notes architecture this Connector requires native libraries (for both session types: IIOP as well as LocalClient), and is therefore only supported on Windows platforms; and the path to the local client libraries as well your Domino server installation should be added to the definition of the PATH variable in the Tivoli Directory Integrator Server startup script, `ibmdisrv.bat`.
2. Refer to Supported session types by Connector for an overview of which session types are possible with this Connector.

When running the Connector reports the object changes necessary to synchronize other repositories with a Domino database, regardless of whether these changes have occurred while the Connector has been offline or they are happening while it runs.

The Domino Change Detection Connector operates in Iterator mode, and reports document changes at the Entry level only.

On each AssemblyLine iteration the Domino Change Detection Connector delivers a single document object which has changed in the Domino database. The Connector delivers the changed Domino document objects as they are, with all their current items and also reports the type of object change - whether the document was added, modified or deleted. The Connector does not report which items have changed in this document or the type of item change. After the Connector retrieves a document change, it parses it and copies all the document items to a new Entry object as Entry Attributes. This Entry object is then returned by the Connector.

This connector supports Delta Tagging at the Entry level only.

This Connector can be used in conjunction with the IBM Password Synchronization plug-ins. For more information about installing and configuring the IBM Password Synchronization plug-ins, see the *IBM Tivoli Directory Integrator V7.1 Password Synchronization Plug-ins Guide*.

The Connector stores locally, on the IBM Tivoli Directory Integrator 7.1 machine, the state of the synchronization. When started it continues from the last synchronization point and reports all changes after this point, including these changes that happened while the Connector was offline.

Note: Changed documents are not delivered in chronological order or in any other particular order, unless you check the "Deliver Sorted" checkbox in the configuration screen. Refer to “Sorting” on page 52 for more information. Without using this option, it means that documents changed later can be delivered before documents changed earlier and vice versa.

The Connector will signal end of data and stop when there are no more changes to report. It can however be configured not to exit when all changes have been reported, but stay alive and repeatedly poll Domino for changes.

Using the Connector

Document identification: The Domino Change Detection Connector retrieves the Universal ID (UnID) of Domino documents. Use the UnID value to track document changes reported by the Connector.

For example, when a deleted document is reported, use its UnID value to lookup the object that has to be deleted in the repository you are synchronizing with. If you are synchronizing Domino users (Person documents), then you might need to find out when a user is renamed. When a user is renamed (the

FullName item of the Person document is changed), the Connector will report this as a "modify" operation. When you lookup objects in the other repository by UnID, you will be able to find the original object, read its old FullName attribute, compare it against the new FullName value and determine that the user has been renamed.

Deleted documents: Documents that are deleted from a Domino database can be tracked by "deletion stub" objects. Deletion stubs provide the Universal ID and Note ID of the deleted document, but nothing more. That is why when the Connector comes across a deleted document, it returns an Entry which does not contain any document items, but only the following Entry Attributes added by the Connector itself:

- "$$UNID"
- "$$NoteID"
- "$$ChangeType"

Minimal synchronization interval: There is a parameter for each database called "Remove documents not modified in the last x days". Deletion stubs older than this value will be removed. If you are interested in processing deleted documents, you must synchronize (run the Connector) on intervals shorter than the value of this parameter.

On both Domino R8.0 and Domino R8.5, this parameter can be accessed from the Lotus Domino Administrator: open the database, then choose from the menu **File -> Replication -> Options for this Application...**, select **Space Savers** – the parameter is called **Remove documents not modified in the last x days**.

The default value of this parameter is 90 days.

Switching to a database replica: UnIDs are the same across replicas of the same database. This allows you to switch to another replica of the Domino database in case the original database is corrupted or not available.

Document timestamps, however, are different for the different replicas. That is why when a switch to a replica is done, you must perform a full synchronization (use a new key for "Iterator State Key" and set the "Start At" parameter to "Start Of Data"). This will possibly report a lot of document additions and deletions which have already been applied to the other repository, but will guarantee that no updates are missed.

Structure of the Entries returned by the Connector: All items contained in a document are mapped to Entry Attributes with their original item names.

All date values are returned as java.util.Date objects.

The following Entry Attributes are added by the Connector itself (their values are not available as document items):

- $$UNID – the Universal ID of the document (see "The $$UNID and $$NoteID Attributes")
- $$NoteID – the Note ID of the document (see "The $$UNID and $$NoteID Attributes")
- $$ChangeType – the type of document modification (see "The $$ChangeType Attribute")
- $$DateCreated – a java.util.Date object representing the time this document was created (this Attribute is available for non-deleted documents only).
- $$DateModified - a java.util.Date object representing the time of the last modification of this document (this Attribute is available for non-deleted documents only).

The $$UNID and $$NoteID Attributes: The Universal ID (UnID) is the value that uniquely identifies a Domino document. All replicas of the document have the same UnID and the UnID is not changed when the document is modified. This value should be used for tracking objects during synchronization. The

Universal ID value is mapped to the $$UNID Attribute of Entry objects delivered by the Connector. The value of the $$UNID Attribute is a string of 32 characters, each one representing a hexadecimal digit (0-9, A-F).

The Connector also returns the NoteID document values. This value is unique only in the context of the current database (a replica of this document will in general have a different NoteID). The Connector delivers the NoteID through the $$NoteID Entry's Attribute. The value of this Attribute is a string containing up to 8 hexadecimal characters.

The $$ChangeType Attribute: An Attribute named $$ChangeType is added to all Entries returned by the Domino Change Detection Connector.

The value of the $$ChangeType Attribute can be one of:

- **add** – means that the document reported is a newly added document in the Domino database
- **modify** – means that the document reported is an already existing document that has been modified
- **delete** – means that the document reported has been deleted from the Domino database

Synchronization state values: Several values are saved into the System Store and represent the current synchronization state. The Connector reads these values on startup and continues reporting changes from the right place.

Regardless of the mode in which the Connector is run two synchronization state values are stored in the User Property Store. These two values a stored in an Entry object as Attributes with the following names and meaning:

- ***SYNC_TIME*** – this Attribute is a java.util.Date object representing the "since" value for the next poll of the Connector, that is, the next Connector's poll will return only database modifications that occurred at or after this time. In the special case when *Start Of Data* is used as a start condition, the java.lang.String value "NULL_DATE" is stored.
- ***SYNC_CHECK_DOCS*** – this Attribute is a java.lang.Boolean object, which indicates whether the Connector must check for already processed documents in the Connector-specific System Store table (see below). This Attribute is only used when the Connector State Key Persistence parameter is set to *After read*. When the Connector State Key Persistence parameter is set to *End of cycle* the value of this Attribute is always **false**.

When the Connector is run, in addition to storing values in the User Property Store it creates (if not already created) a Connector-specific table in the System Store. The name of this table is the concatenation of "***domch_***" and the value of the ***Iterator State Key*** Connector parameter. This Connector-specific table stores values with the following characteristics:

- The keys are the UnIDs of already delivered changed documents as java.lang.String objects
- The values are java.util.Date object representing the datetime for the next poll as it was at the time this document was delivered by the Connector; if however the UnID corresponds to a deleted document, the java.lang.String constant "NULL_DATE" is stored instead.

The Connector-specific table is cleared each time the Connector successfully completes a synchronization session.

For each instance of the Domino Change Detection Connector executed on the same IBM Tivoli Directory Integrator Server there is a different Connector-specific table in the System Store.

Accessing the Connector synchronization state: While the Connector is offline you can access the "since" datetime that will be used on the next Connector run. This datetime is stored in the User Property Store.

This is how you can get the datetime value for the next synchronization:

```
var syncTime = system.getPersistentObject("dcd_sync");
var sinceDateTimeAttribute = syncTime.getAttribute("SYNC_TIME");
var sinceDateTime = sinceDateTimeAttribute.getValue(0);
if (sinceDateTime.getClass().getName().equals("java.util.Date")) {
main.logmsg("Start date: " + sinceDateTime);
}
else {
main.logmsg("Start date: Start Of Data");
}
```

"dcd_sync" is the value specified by the **Iterator Store Key** Connector parameter.

This is how you can set a start datetime for the next synchronization:

```
var syncTime = system.newEntry();
syncTime.setAttribute("SYNC_TIME", new java.util.Date()); //current time
syncTime.setAttribute("SYNC_CHECK_DOCS", new java.lang.Boolean("false"));
system.setPersistentObject("dcd_sync", syncTime);
```

Filtering entries: No filtering of documents is performed in this version of the Connector. All database documents that have been created, modified or deleted are reported by the Connector.

If you need filtering you must do this yourself by scripting in the Connector hooks.

Sorting: The changed documents can be delivered sorted by the date they were last modified on. This is done by checking the checkbox "Deliver Sorted" in the configuration screen.

Note: Using sorting comes with a performance penalty, in terms of memory usage and CPU time. That is why you should consider carefully whether you really need sorting.

Domino Server system time is used: The Domino Change Detection Connector uses the timestamp of last modification for detecting changes in a Domino database. The Connector state includes timestamp values read by the Domino Server system clock. That is why changing the Domino Server system time while the Connector is running or between Connector runs might result in incorrect Connector operation – changes missed or repeated, incorrect change type reported, etc.

Processing very large Domino databases (.nsf files): The Connector could need a bigger amount of physical memory – for example, when working on very large databases containing 1,000,000 documents or more, especially when performing a full synchronization. This is caused by the Connector keeping all retrieved document UnIDs in memory for the duration of the synchronization session. For example, 512 MB of physical memory should be enough for processing a database that contains about 1,000,000 changed documents (provided that no other memory consuming processes are running). If this amount of memory is unavailable, then you can increase the memory available to IBM Tivoli Directory Integrator.

Also, be mindful of the "Deliver Sorted" parameter - enabling this could have a major performance impact.

API: The Domino Change Detection Connector supports two methods specific to it, as follows:

```
    /**
     * This method saves the synchronization state of the Connector.
     * This method should be called by users (using script component) whenever
     * they want to save the synchronization state.
     *
     * @throws Exception if the synchronization fails.
     */
 public void saveStateKey() throws Exception;

 /**
  * Skip the current document. Use this method to skip problem documents when
  * the Connector will otherwise die with an exception.
  * <p>
```

```
 * For example use the following script in the "Default On Error" hook of
 * the Connector:
 *
 * <pre>
 * thisConnector.connector.skipCurrentDocument();
 * </pre>
 *
 * </p>
 *
 *
 * @throws Exception
 *             If the Notes thread is not running or the Notes thread
 *             encounters an error while processing the command.
 */
public void skipCurrentDocument()throws Exception;
```

Required Setup of the IBM Tivoli Directory Integrator

See the section, Supported session types by Connector and the sections below about the issue of required libraries, and possible library conflicts.

Required Domino Setup

Required Domino Server tasks: The Connector requires that the following Domino Server tasks be started on the Domino Server:

- HTTP Web Server
- DIIOP Server

If these Domino Server tasks are not started on the server the Connector will fail.

Required privileges: The Domino Change Detection Connector creates two sessions to the Domino Server – a session through the local Notes client code using the local User ID file and a remote IIOP session using an internet user account (the same Domino user can be used for establishing both sessions but this is not required). The accounts used for these sessions must have the following privileges:

The account of the local User ID

The Connector uses the Notes client User ID file for connecting to the Domino Server. That is why the Connector needs the Notes client User ID file to be set up properly for accessing the Domino Server. The Domino user whose User ID file is deployed locally needs at least the "Reader" Access configured in the Access Control List (ACL) of the Domino database that is polled for changes.

You can configure this from the "Files" tab of the Lotus Domino Administrator: right click on the database which will be polled for changes, select "Access Control -> Manage...". If you don't see the user name associated with this User ID file listed, click the "Add..." button and add this user name to the list. Select this user name in the list and make sure that the Access is set to "Reader" or higher (that is, "Reader", "Author", "Editor", "Designer" or "Manager") for this user.

The internet account for the IIOP session

The Connector needs the username and password of a Lotus Domino Internet user for creating the IIOP session. The Internet user must have at least the "Reader" Access configured in the Access Control List (ACL) of the Domino database that is polled for changes.

You can configure this from the "Files" tab of the Lotus Domino Administrator: right click on the database which will be polled for changes, select "Access Control -> Manage...". If you don't see the Internet user listed, click the "Add..." button and add the Internet user to the list. Select the Internet user name in the list and make sure that the Access is set to "Reader" or higher (that is, "Reader", "Author", "Editor", "Designer" or "Manager") for this user.

Configuration

The Domino Change Detection Connector provides the following parameters:

Session Type
Specifies whether the Connector will create an IIOP session or performs LocalClient calls. This is a drop-down list; the default value is "IIOP".

For **Session Type=LocalClient**, the following parameters are disregarded: **IOR String**, **HTTP Port**, **Username** and **Use SSL**.

Domino Server IP Address
The IP address (or hostname) of the Domino Server where the database that will be polled for changes resides.

IOR String
The IOR string used to create the IIOP session. This parameter can optionally be used instead of requesting this value from the Domino server.

HTTP Port
The port on which the HTTP task of the Domino Server is running. The default value is 80.

Username
User name for IIOP session authentication. This must match the first value of the "User name" field of that user's Person document.

Password
User password for IIOP session authentication. This must match the "Internet Password" field of the user's Person document.

Use SSL
Enables encrypted communications with the Domino server, using client-side certificates. The parameter is relevant only for IIOP Sessions.

Database
The filename of the Domino database which will be polled for changes, for example "names.nsf".

Iterator State Key
Specifies the name of the parameter that stores the current synchronization state (i.e., the last change) in the User Property Store of the IBM Tivoli Directory Integrator. This should be a unique name for all parameters stored in one instance of the IBM Tivoli Directory Integrator's User Property Store.

The **Delete** button clears all synchronization state associated with the value of this parameter. When clicked, the Connector deletes the key-value pair from the User Property Store as well as the Connector-specific table from the System Store.

Start at
The type of starting condition. This parameter is taken into account only when the persistent parameter specified by **Iterator State Key** is blank or not found in the System Store. Can be one of:

Start Of Data
Performs a full synchronization retrieving all documents from the database.

End Of Data
Retrieve future changes only (changes that are done after the Connector is started.)

Specific date
Retrieve changes that occurred at or after the value specified by the **Start Date** parameter.

The default value is "**Start Of Data**".

Note: This parameter is taken into account only when the persistent parameter specified by **Iterator State Key** is not found in the User Property Store.

Start Date
The Connector will retrieve documents which have been changed at or after this date/time. This parameter is only used when the **Start At** parameter is set to **Specific Date**, and accepts the following date/time formats:

- **yyyy-MM-dd HH:mm:ss.SSS —** for example: 2002-05-23 16:39:07.628 (that is 4-digit year, 2-digit month, 2-digit day, 2-digit 24-hour hour, 2-digit minutes, 2-digit seconds and 3-digit milliseconds). Please note that the actual precision of Lotus Notes date/times is 10 milliseconds.
- **yyyy-MM-dd HH:mm:ss —** for example: 2002-05-23 16:39:07
- **yyyy-MM-dd —** for example: 2002-05-23

It is only taken into account when the persistent parameter specified by "**Iterator State Key**" is blank or not found in the System Store and the "**Start At**" parameter is set to "**Specific Date**".

State Key Persistence
Governs the method used for saving the Connector's state to the System Store. The default is **End of Cycle**, and choices are:

After Read
Updates the System Store when you read an entry from the Domino change log, just before you continue with the rest of the AssemblyLine. This mode of operation was called "Assured once and only once delivery" in older versions of Tivoli Directory Integrator.

End of Cycle
Updates the System Store when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual
Switches off the automatic updating of the System Store with this Connector's state information; instead, you will need to save the state by manually calling the Domino Change Detection Connector's *saveStateKey()* method, at a suitable place at your discretion in your AssemblyLine.

Timeout
Specifies the maximum number of seconds the Connector waits for the next changed document. If this parameter is 0, then the Connector waits forever. If the Connector has not retrieved the next changed document object within timeout seconds from the beginning of the waiting, then it returns a NULL Entry, indicating that there are no more Entries to return.

Sleep Interval
Specifies the number of seconds the Connector sleeps between successive polls for changes.

Deliver Sorted
If checked, the changed documents are delivered sorted by the date they were last modified on; otherwise the changed documents are delivered in random order. If the number of changed documents is large then sorting could slow Tivoli Directory Integrator performance.

Detailed Log
Check to enable additional log messages.

Troubleshooting the Domino Change Detection Connector

1. **Problem:** When you run an AssemblyLine that uses the Domino Change Detection Connector, the AssemblyLine fails with the following exception: *NotesException: Could not get IOR from Domino Server:* ... where *<domino_server_ip>* is the IP address of the Domino Server you are trying to access, that is, the value of the **Domino Server IP address** Connector parameter.

   Solution: This exception indicates that the HTTP Web Server task on the Domino Server is not running. Start the HTTP Web Server task on the Domino Server you are trying to access and then start your AssemblyLine again.

2. **Problem:** When you run an AssemblyLine that uses the Domino Change Detection Connector, just after you enter the User ID password at the password prompt the AssemblyLine fails with the following exception: *NotesException: Could not open Notes session: org.omg.CORBA.COMM_FAILURE: java.net.ConnectException: Connection refused: connect Host: <domino_server_ip> Port: XXXXX vmcid: 0x0 minor code: 1 completed: No* where `<domino_server_ip>` is the IP address of the Domino Server you are trying to access, that is, the value of the **Domino Server IP address** Connector parameter.

   Solution: This exception indicates that the DIIOP Server task on the Domino Server is not running. Start the DIIOP Server task on the Domino Server you are trying to access and then start your AssemblyLine again.

   Another reason for this message is that the fully qualified host name of the Lotus Domino server is not correctly set (for example, it was left as localhost or 127.0.0.1). To solve this problem start "Domino Admin", open the server used, go to the **Configuration** tab and edit the "Server->Current Server document". In this document under the **Basic** tab you must add the correct value for "Fully qualified Internet host name", save the document and restart the server.

3. **Problem:** While the Domino Change Detection Connector is retrieving changes the following exception occurs: *Exception in thread "main" java.lang.OutOfMemoryError*

   Solution: This exception indicates that the memory available to the IBM Tivoli Directory Integrator Java Virtual Machine (the JVM maximum heap size) is insufficient. In general the Java Virtual Machine does not use all the available memory. You can increase the memory available to the IBM Tivoli Directory Integrator JVM by following this procedure:

   Edit *ibmdisrv.bat* in the IBM Tivoli Directory Integrator install directory to adjust the existing -Xms16m option to -Xms254m -Xmx1024m in the next to last line of the file (that is, the line that invokes Java).

   Note: -Xms is the initial heap size in bytes and -Xmx is the maximum heap size in bytes. You can set these values according to your needs.

4. **Problem:** The Connector reports all database documents as ***deleted*** although they are not deleted.

   Solution: The user of the local User ID file is not given the necessary privileges on the database polled for changes. Give the necessary user rights as described in "Required privileges" on page 53.

5. **Problem:** When you run an AssemblyLine that uses the Domino Change Detection Connector, the following exception occurs: *java.lang.UnsatisfiedLinkError: <Tivoli Directory Integrator_install_folder>\libs\domchdet.dll: Can't find dependent libraries* where *<Tivoli Directory Integrator_install_folder>* is the folder where IBM Tivoli Directory Integrator is installed.

   Note: If you run the Tivoli Directory Integrator Server from the command prompt, then before this exception message is printed, a popup dialog box appears saying "This application has failed to start because nNOTES.dll was not found. Re-installing the application may fix this problem."

   Solution: This exception message as well as the popup dialog box are displayed because the Connector is unable to locate the Lotus Notes dynamic-link libraries. Most probably the path to the Lotus Notes directory specified in ibmditk.bat or in ibmdisrv.bat is either incorrect or not specified at all. That is why you should verify that the Lotus Notes directory specified in the PATH environment variable in both ibmditk.bat and ibmdisrv.bat is correct. For more information please see "Required Setup of the IBM Tivoli Directory Integrator" on page 53.

6. **Problem:** Some of the documents contain invalid data and cause the Connector to throw an exception and stop. These documents comprise only a small fraction of the whole database. Skip the problem documents and continue iterating.

   Solution: Override the "Default On Error" hook of the Connector. Use the `skipCurrentDocument()` method to increment the internal document counter of the Connector so that it skips the problem document. Also use the `system.skipEntry()` method to instruct the AssemblyLine to skip the current cycle – the Connector failed to read the document so it has no meaningful data to provide for this cycle.

   The script that you put in the "Default On Error"hook should make difference between non-fatal errors (for example, document contains an invalid field) and fatal errors (for example, the Domino

server is not running). You should not let the Connector continue iterating after a fatal error occurs. Here is a sample script for the "Default On Error" hook. The script skips only documents which contain an invalid date:

```
var ex = error.getObject("exception");
var goOn = false;
if (ex != null) {
 if (ex.getMessage().indexOf("Invalid date") != -1) {
  goOn = true;
 }
}
if (goOn) {
 thisConnector.connector.skipCurrentDocument();
 system.skipEntry();
} else {
 throw "Fatal error: "+error;
}
```

7. **Problem:** When you run an AssemblyLine that uses the Domino Change Detection Connector, the AssemblyLine fails with the following exception:

   ```
   java.lang.Exception: CTGDJE010E Connector was unable to initialize local Notes session to Domino Server.
   Exception encountered: java.lang.Exception: Native call SECKFMSwitchToIDFile failed with error: code 259, 'File does not exist'.
   ```

   Solution: Detailed information for this problem can be found in section “Post Install Configuration” on page 45, both in paragraphs "If you create Local Client Session" and "If you create IIOP Session".

Compatibilty: Refer to the section on Supported session types by Connector on how this connector should be set up with the necessary libraries, and about interactions with other Domino/Lotus Notes Connectors.

See also

Accessing Java Session Objects,
Accessing documents using Java classes.

Domino Users Connector

The Domino Users Connector provides access to Lotus Domino user accounts and the means for managing them. With it, you can do the following actions:

- Retrieve users documents and their items from the Name and Address Book
- Create and register Domino users
- Initiate Domino users deletion (through the Domino Administration Process) by posting administration requests to the Administration Requests Database
- Modify users by modifying their Person documents in the Name and Address Book
- Perform users' "disabling/enabling" by adding/removing users' names to/from a "Deny Access Group"
- Perform "lookup" of Domino users.

Currently, the Connector does not support the process of Users recertifying.

The Domino Server accessed can be on a remote server, or on the local machine.

It operates in Iterator, Lookup, AddOnly, Update and Delete modes, and enables the following operations to be performed:

Iterator

Iterate over all (or a filtered subset of) Person documents from the Name and Address Book.

The Connector iterates through the Person documents of the 'Name and Address Book' database. All Person documents (matching the filter, if filter is set) are delivered as Entry objects, and all document items, except attachments, are transformed into Entry Attributes.

Along with the Attributes corresponding to the Person document items, the Entry returned by the Connector will contain some extra Attributes, created by the Connector itself. The table below describes these Attributes. Their names will be prefixed with "DER_" to indicate that they have been derived by the Connector, and they are not "native" Domino Attributes):

Table 6. Derived Attributes

Attribute Name	Type	Value
DER_IsEnabled	Boolean	*true* – if the user does not belong to a "Deny List only" group; *false* – if the user belongs to at least one group of type "Deny List only".

Lookup

Search for and retrieve Person documents that match some criteria.

In Lookup mode, the Connector will perform different type of searches, depending on the value of the **Use full-text search** parameter:

- **Use full-text search = "true"**: The Connector will perform a full-text search in the "People" view.

  Note: "Full-text search" will work both with full-text indexed and not full-text indexed databases; however, the search will be less efficient if the database is not full-text indexed.

  It is also possible that the database full-text index will not be updated, in which case the search results would not match the actual database content.
- **Use full-text search = "false"**: the Connector will perform a regular database search using Lotus formula. The element (Form = "Person") will be automatically added to the formula by the Connector, so the search will be limited to user documents only.

AddOnly

Register new users in Domino Server and create their Person documents. When doing so, you

have the option to specify a mail template when registering users. If a template is not specified the Connector will continue to work as the TDI 6.0 version of the Connector (that is, use the default template).

Update
Modify users' Person documents; Enable/disable users; Register existing (internet) users, as well as "disabling/enabling" by adding/removing users' names to/from a "Deny Access Group".

Delete Post requests for user deletion in the Domino Server Administration Requests Database.

This Connector can be used in conjunction with the IBM Password Synchronization plug-ins. For more information about installing and configuring the IBM Password Synchronization plug-ins, please see the *IBM Tivoli Directory Integrator V7.1 Password Synchronization Plug-ins Guide*.

Note: The Domino Users Connector requires Lotus Notes to be release 7.0 or 8.0; and Lotus Domino Server version 7.0 and 8.0.

Deployment and connection to Domino server

Refer to the section, Supported session types by Connector for more information about required libraries setup, and possible library conflicts.

Deploying on a Domino Server machine
When the Domino Users Connector is deployed on a machine where a Domino Server is installed you can use both Authentication mechanisms supported – Internet Password authentication and Notes ID File authentication.

Deploying on a Notes client machine
When the Domino Users Connector is deployed on a machine where a Notes client is installed you can only use Notes ID File authentication.

To authenticate the local server connection, Domino requires the user's short name and internet password (these are Connector's parameters).

Configuration

The Connector needs the following parameters:

Session Type
Specifies whether the Connector will create an IIOP session or performs LocalClient calls. This is a drop-down list; the default value is "IIOP".

For **Session Type=LocalClient**, the following parameters are disregarded: **IOR String**, **HTTP Port**, **Username** and **Use SSL**. Also see “Parameter migration from earlier versions” on page 61 for more information.

Domino Server IP Address
The IP address (or hostname) of the Domino Server, which hosts the 'Name and Address Book' Database.

If this parameter is missing or empty, the local machine is used. This behavior ensures backward compatibility with pre-6.1 TDI configuration files.

IOR String
The IOR string used to create the IIOP session. This parameter can optionally be used instead of requesting this value from the Domino server.

HTTP Port
The port on which the HTTP task of the Domino Server is running. The default value is 80.

Username
The user name used for log in or authentication to the Domino Server. Ignored if **Session Type=LocalClient** authentication is used. See “Authentication” on page 62 for more details.

Password
The password for the **Username**, or password associated with the Notes ID File if that type of authentication is used. See “Authentication” on page 62 for more details.

Use SSL
Enables encrypted communications with the Domino server, using client-side certificates. The parameter is relevant only for IIOP Sessions.

Name and Address Book Database
The name of the Domino Directory database (previously known as the "Name and Address Book" database). Usually it is "names.nsf" (which is the default.)

Use full-text search
This parameter is used when the Connector is configured in Iterator or Lookup modes. If checked, the Connector accesses user documents through the **People** view and full-text searches. If not checked, the Connector uses regular database searches. In this case the Connector automatically narrows the database search to user documents only, by accessing only documents for which **Form item** value is **Person**. This parameter affects the Iterator and Lookup modes only.

Full-text filter
This value is taken into account only when **Use full-text search** is enabled. This parameter contains full-text query that filters the user documents returned by the Connector in Iterator mode. If null or empty string, no filtering is performed. The default value is empty.

Formula filter
This value is taken into account only when **Use full-text search** is not enabled. This parameter contains a formula that filters the users returned by the Connector in Iterator mode. The Connector automatically adds the following snippet to this formula:

```
"& Form = "Person""
```

which limits the search to user documents only. The default value is empty.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Parameter migration from earlier versions

Tivoli Directory Integrator 7.1 introduces a **Session Type** parameter for this Connector, as part of a harmonization with the other Lotus Notes/Domino Connectors. The **Session Type** parameter covers functionality previously configured through the Authentication Mechanism parameter:

- If the **Session Type** parameter exists in the Config file, its value will be always used no matter if the **Authentication Mechanism** parameter exists or not.
- However, if the **Session Type** parameter is not used but the **Authentication Mechanism** parameter is present then a mapping is done to guarantee backward compatibility:
  - **Notes ID File** is mapped to "LocalClient",
  - **Internet Password** is mapped to "LocalServer".

Security

To have the IBM Tivoli Directory Integrator access the Domino Server, you might have to enable it through **Domino Administrator -> Configuration -> Current Server Document -> Security -> Java/COM Restrictions**. The user account you have configured the IBM Tivoli Directory Integrator to use must belong to a group listed under **Run restricted Java/Javascript/COM** and **Run unrestricted Java/Javascript/COM**.

Configuring encryption between the Domino Server and a client: When the Domino Users Connector is running on a Notes client machine, there is communication going on between the Notes client machine and the Domino Server machine.

Port encryption in Domino and/or Notes can be used to encrypt this communication. Two options are available:

Encrypt Domino Server communication ports

This is easier to setup (the Server settings only are configured), but this affects the communication with all clients including regular users using Lotus Notes clients.

1. In Lotus Domino Administrator select **Configuration**.
2. Select **Server/Server Ports...** from the right-side panel.
3. For each communication port in use, select the port in the **Communication ports** list and check the **Encrypt network data** option.
4. Click **OK**.
5. Restart the Domino Server for changes to take effect.

Encrypt Lotus Notes communication ports

This does not affect other Notes clients if encryption is not necessary for them.

1. In Lotus Notes go to **File->Preferences->User Preferences...**
2. Select **Ports** from the left navigation panel.
3. For each communication port in use, select the port in the **Communication ports** list and check the Encrypt network data option.
4. Click **OK**.
5. Restart Lotus Notes for changes to take effect.

Authentication: The Domino Users Connector impersonates as a Domino user in order to access the Domino Directory (Names and Address Book database).

The Domino Users Connector supports two authentication mechanisms – Internet Password authentication and Notes ID file based authentication.

Internet Password Authentication – used with IIOP and Local Server sessions

This authentication mechanism uses the Domino user's Short Name and Internet password. The Domino user's Short Name and Internet password must be supplied as Connector configuration parameters **Username** and **Password**.

The Domino Users Connector uses this mechanism in order to create an Internet Session object for making local calls based on the Domino Directory. This authentication mechanism requires that a Domino Server is installed on the local machine.

Notes ID File Authentication – used with Local Client session

This authentication mechanism uses the currently configured default Notes ID file along with its password. A local client session is created using the password parameter. Access is granted if the value of this parameter matches the Notes user ID.

The currently configured default Notes ID file is a part of the Notes client configuration. Normally the Notes client stores the path to the currently configured default Notes ID file in the `notes.ini` file, so that when the next time Notes client starts it will use this Notes ID file by default.

The password of the Notes ID file must be supplied as a Connector configuration parameter **Password**.

The Domino Users Connector uses this authentication mechanism in order to create a Session object for making local calls based on the Notes user ID. A Domino server or Notes client must be installed locally.

This authentication mechanism can be used both on a Notes client machine and on a Domino Server machine. When this mechanism is used on a Domino Server machine the Server ID file is used. Normally Server ID files do not have passwords or have empty passwords; that is why you

would normally leave the **Password** Connector configuration parameter blank. If, however, the Server ID file does have a password you should specify that password as the value for the **Password** Connector configuration parameter.

Authorization: The Domino Server uses the Access Control Lists of the Domino Directory (Names and Address Book database) to verify that the Domino user which the Connector uses has actually the right to access the required database, document or field.

If the Connector is used to change the FirstName or LastName or both of a Domino user, then the Access Control Lists of databases to which the user used to have access before the renaming occurred must be updated manually, so that the new user name would be recognized.

Using the Domino Users Connector

Iterator mode: The Connector iterates through the Person documents of the **Name and Address Book** database. All Person documents (matching the filter, if filter is set) are delivered as Entry objects, and all document items, except attachments, are transformed into Entry attributes.

Along with the attributes corresponding to the Person document items, the Entry returned by the Connector contains some extra (derived) attributes for which values are calculated by the Connector. Here is the list of the derived attributes:

DER_IsEnabled
(Boolean) Specifies whether the user is enabled/disabled:

- **true** - if the user does not belong to a **Deny List only** group
- **false** - if the user belongs to at least one **Deny List only** group

Lookup mode: In Lookup mode, the Connector performs searches for user documents, and the type of search depends on the value of the **Use full-text search** parameter:

- **Use full-text search = true**: The Connector performs a full-text search in the People view. Full-text searches work both with full-text indexed and not full-text indexed databases. However, the search is less efficient if the database is not full-text indexed. It is also possible that the database full-text index is not updated, in which case the search results do not match the actual database content.
- **Use full-text search = false**: The Connector performs a regular database search using Lotus formula. The element (Form = "Person") is automatically added to the formula by the Connector, so the search is limited to user documents only.

When simple link criteria are used, you can use both canonical (CN=UserName/O=Org) and abbreviated (UserName/Org) name values to specify the user's FullName. The Connector automatically processes and converts the value you specified, if necessary.

When advanced link criteria is used, you must be careful and specify the user's FullName in the correct format, which is:

- for full-text search: use abbreviated names (UserName/Org)
- for regular database search: use canonical names (CN=UserName/O=Org)

AddOnly mode: The AddOnly mode always adds a new Person document in the **Name and Address Book** database. The add process accepts whatever attributes are provided by the Attribute Mapping, however to have correct user processing by Domino, the attribute names must match the **Item** names Domino operates with. As the Connector operates with users only, it always sets the attributes **Type** and **Form** to the value of **Person**, thus overriding any values set to these attributes during the Attribute Mapping process. The **LastName** Domino user attribute is required for successful creation of a Person document. The HTTPPassword attribute is not required, but if present its value is automatically hashed by the Connector.

Depending on a fixed schema of attributes, the Connector can register the new user. The table below specifies these attributes and the Connector behavior according to their presence or absence in the *conn* Entry, and their values:

Attribute name	Type	Required for registration?	Value
REG_Perform	Boolean	Yes	If set to true the Connector performs user registration. If this attribute is missing, or its value is false, the Connector does not perform user registration, regardless of the presence and the values of the other REG_ Attributes.
REG_IdFile	String	Yes	Contains the full path of the ID file to be registered. For example, `c:\\newuserdata \\newuser.id`
REG_UserPw	String	No	The user's password.
REG_Server	String	No	The name of the server containing the user's mail file. If the Attribute is missing, the value will be obtained from the current Connector's Domino Session. When the Connector is running on a Notes client machine and is registering a user, this Attribute must be specified in order to create a mail file on the server for the newly registered user.
REG_CertifierIDFile	String	Yes	The full file path to the certifier ID file.
REG_CertPassword	String	Yes	The password for the certifier ID file. **Note:** If the certifier password is wrong when registering users, a popup window is displayed. Ensure that the Certifier password is correctly specified.
REG_Forward	String	No	The forwarding domain for the user's mail file.
REG_AltOrgUnit	Vector of <String>	No	Alternate names for the organizational unit to use when creating ID file.
REG_AltOrgUnit Lang	Vector of <String>	No	Alternate language names for the organizational unit to use when creating ID file.
REG_CreateMailDb	Boolean/String	No	true – Creates a mail database; false – Does not create a mail database; it is created during setup. If this attribute is missing, a default value of false is assumed. If this attribute is true, the MailFile attribute must be mapped to a valid path.
REG_MailTemplateFile	String	No	The filename of a Notes template database, which the Connector will use to create the user mail file. If this Attribute does not exist the default mail template is used.

Attribute name	Type	Required for registration?	Value
REG_MailTemplateServer	String	No	The IP address or hostname of the Domino server machine on which the mail template database (specified by "REG_MailTemplateFile") resides. If this Attribute does not exist the local Domino server machine is used.
REG_MailDbInherit	Boolean/String	No	true – the user mail database to be created will inherit any changes to the mail template database design; false – the user mail database to be created will **not** inherit any changes to the mail template database design. If this Attribute is missing, a default value of "false" will be assumed.
REG_StoreIDInAddress Book	Boolean/String	No	true – stores the ID file in the server's Domino Directory; false – does not store the ID file in the server's Domino Directory. If this Attribute is missing, a default value of "false" is used.
REG_Expiration	Date	No	The expiration date to use when creating the ID file. If the attribute is missing, or its value is null, a default value of the current date + 2 years is used.
REG_IDType	Integer/String	No	The type of ID file to create: 0 - create a flat ID; 1 - create a hierarchical ID; 2 - create an ID that depends on whether the certifier ID is flat or hierarchical. If the attribute is missing, a default value of 2 is used.
REG_Is NorthAmerican	Boolean/String	No	true – the ID file is North American; false – the ID file is not North American. If this attribute is missing, a default value of true is used.
REG_MinPassword Length	Integer/String	No	The REG_MinPasswordLength value defines the minimum password length required for subsequent changes to the password by the user. The password used when the user registers is not restricted by the REG_MinPasswordLength value. If this attribute is missing, a default value of 0 is used.
REG_OrgUnit	String	No	The organizational unit to use when creating the ID file. If this attribute is missing, a default value of " " is used.
REG_RegistrationLog	String	No	The log file to use when creating the ID file. If this attribute is missing, a default value of " " is used.

Attribute name	Type	Required for registration?	Value
REG_Registration Server	String	No	The server to use when creating the ID file. This attribute is used only when the created ID is stored in the server Domino Directory, or when a mail database is created for the new user.
REG_StoreID InAddressBook	Boolean/String	No	true - stores the ID file in the server's Domino Directory false - does not store the ID file in the server's Domino Directory. If this attribute is missing, a default value of false is used.

The attributes for which the **Required for registration** field is set to **Yes** are required for successful user registration. Along with these REG_ Attributes, the **LastName** Domino user attribute is also required for successful user registration.

If REG_Perform is set to **true** and any of the other attributes required for registration are missing, the Connector throws an Exception with a message explaining the problem.

Update mode: In Update mode, the following happens:

1. A search for the Entry to be updated is performed in Domino.
2. If an Entry is not found, an AddOnly operation is performed as described in the AddOnly mode (including user registration if the necessary REG_ Attributes are supplied).
3. If the Entry is found, a modify operation is performed.

When modifying a user, the Domino Users Connector always modifies its Person document in the **Name and Address Book** database with the attributes provided. The modify process accepts whatever Attributes are provided by the Attribute Mapping, however to have correct user processing by Domino, the Attribute names must match the **Item** names Domino operates with. See "List of Domino user attributes (or Person document items)" on page 69 for a (possibly not full) list of Domino user properties.

As the Connector operates with users only, it does not modify the attributes **Type** and **Form** (their value must be **Person**) regardless of the Attribute Mapping process. If the **HTTPPassword** attribute is specified, its value is automatically hashed by the Connector.

In the process of modifying users, the Domino Users Connector provides the options to disable and enable users. A user is disabled by adding his name into a specified **Deny List only** group (consult the Domino documentation for information on **Deny List only** groups. Go to http://www.lotus.com/products/domdoc.nsf, and click the **Lotus Domino Document Manager 3.5** link). A user is enabled by removing his name from all **Deny List only** groups.

The Connector performs user disabling or enabling depending on the presence in the **conn** Entry, and the values of the following Entry attributes:

ACC_SetType
(Integer/String) If this attribute is missing, no actions are performed and the user keeps its current disable/enable status. If this Attribute is provided, its value is inspected:

- **0** - The Connector performs the operation **disable user** (the user's name is placed in the group specified by the **ACC_DenyGroupName** attribute).
- **1** - The Connector performs the operation **enable user** (any other value results in Exception).

ACC_DenyGroupName
(String) The name of the **Deny List only** group where the user's name is added when disabling the user. When the value of **ACC_SetType** is **0**, the **ACC_DenyGroupName** attribute is required.

If it is missing or its value specifies a non-existing **Deny List only** group, an Exception is thrown. When the **ACC_SetType** attribute is missing, or its value is **1**, the **ACC_DenyGroupName** attribute is not required and its value is ignored.

The Connector can perform user registration on modify too. To determine whether or not to perform registration, the same rules apply as in the AddOnly mode. The same schema of attributes is used and all REG_ Attributes have the same meaning.

If the REG_ Attributes determine that registration is performed, the following cases might happen:

- The user has not yet been registered (for example, this can be an internet or Web user that you want to register and enable to log on and work through a Notes client). The user is then registered, a new ID file is created, and so forth.
- The user has already been registered. In this case the user is re-registered, for example, the Domino registration values are reset with the new values provided. A new ID file is also created.

Notes:

1. When registering users on modify, turn off the **Compute Changes** Connector option. When turned on, the **Compute Changes** function might clear attributes required in certain variants of user registration, and this results in registration failure.
2. When registering users on modify, you must know beforehand what is the user's FullName after registration, and you must provide the attribute **FullName** in the **conn** Entry with this value (which is probably constructed by scripting). This is not very convenient and requires deep knowledge of the Domino registration process. Without setting the expected user's FullName beforehand, however, you risk registering a new user instead of the existing one.
3. When registering users on modify, you must provide the attribute **FirstName** in the **conn** Entry with the value of the FirstName of the user you need to register. If the **FirstName** attribute is not provided, you risk creating a new user.

Delete mode: For user deletion, the Connector uses the Domino Administration Process.

The Connector posts **Delete in Address Book** requests in the **Administration Requests** Database . Each request of type **Delete in Address Book**, when processed by the Domino Administration Process, triggers the whole chain of posting and processing administration requests and actions performed by the Administration Process to delete a user. The result of posting a **Delete in Address Book** administration request is the same as manually deleting a user through the Domino Administrator. In particular:

- The time of processing the administration requests depends on the Domino Server configuration.
- Depending on the type of deletion requested, the chain of administration requests can include requests that require Administrator's approval (for example, the **Approve File Deletion** request for deleting the user's mail file).

The Connector enables tuning of each single user deletion it initiates. The parameters that can be configured are:

Delete mail file

You can specify one of the following options:

- Don't delete mail file.
- Delete just the mail file specified in Person document.
- Delete mail file specified in Person document and all replicas.

Add to group

Specifies if the user's name must be placed in a group when deleting the user, and if **yes**, specifies the name of the group too. This option is usually used to add the user in a **Deny List only** group when deleting the user; thus the user is denied access to the servers.

The delete parameters described previous, have default values that can also be changed through APIs provided by the Domino Users Connector. Each time an instance of the Domino Users Connector is created (in particular on each AssemblyLine start), the parameters have the following default values:

Delete mail file
Don't delete mail file.

Add to group
On deletion, do not add the user's name in any group.

If the default values fit the type of deletion you want, then no special configuration for the deletion is needed. You must specify the correct link criteria in the Delete Connector.

You can however use the APIs provided by the Domino Users Connector, to change these default values at runtime (using scripting):

int getDeleteMailFile()
Returns the code of the default value for the Delete mail file parameter:
- **0** - Don't delete mail file.
- **1** - Delete just the mail file specified in Person document.
- **2** - Delete mail file specified in Person document and all replicas.

void setDeleteMailFile (int deleteType)
Sets the default value for the Delete mail file parameter. The **deleteType** method's parameter must contain the code of the desired value (the codes are as described for getDeleteMailFile()).

String getDeleteGroupName()
Returns the default value for the Add to group parameter:
- NULL - Means **Do not add the user's name in any group**.
- Non-NULL value - The name of the Group where the user's name is added.

void setDeleteGroupName (String groupName)
Sets the default value for the Add to group parameter:
- NULL - Specifies that the user's name must not be added in any group on deletion.
- Non-NULL String value - Specifies the name of the group where the user's name is added on deletion.

The default values for the delete parameters are used in all deletions performed by the Connector, until another change in their values is made, or the Connector instance (object) is destroyed.

The following are possible scenarios that use these methods:
- Script code in the **Before Delete** hook checks the values of the **work** and **conn** objects (and everything else it needs to check), and depending on the specific decision logic uses the **setDeleteMailFile** and **setDeleteGroupName** to tune each particular user deletion.
- If all users for deletion must be deleted using one pattern (and there is no need to tune each particular user deletion), script code in the AssemblyLine Prolog can use the setDeleteMailFile and setDeleteGroupName methods and set the desired values for the whole process.

Another method to manipulate the delete parameters, is to provide the following attributes in the **conn** Entry:

DEL_DeleteMailFile
(Integer/String)

If this attribute is missing in the **conn** Entry, the default value for **Delete mail file** is used.

If this attribute is provided in the **conn** Entry, its value determines the value for the **Delete mail file** parameter for the current deletion only:

- **0** - Don't delete mail file.
- **1** - Delete just the mail file specified in Person document.
- **2** - Delete mail file specified in Person document and all replicas.

DEL_DeleteGroupName
(String)

If this attribute is missing in the **conn** Entry, the default value for **Add to group** is used.

If this attribute is provided in the **conn** Entry, its value determines the value for the **Add to group** parameter for the current deletion only:

- NULL - Specifies that the user's name must not be added in any group.
- Non-NULL String value - Specifies the name of the group where the user's name is added.

The use of the **DEL_DeleteMailFile** and **DEL_DeleteGroupName** attributes in the **conn** Entry overrides the default values of the corresponding delete parameters for the current deletion only.

Setting the **DEL_DeleteMailFile** and **DEL_DeleteGroupName** attributes in the **conn** Entry can be done through scripting in the **Before Delete** hook. Adding attributes by scripting might not be very convenient, so you might prefer to use the default delete parameters values and the APIs that change them.

List of Domino user attributes (or Person document items)

The following is a list (possibly not full) of Domino user document items, which are understood or processed by Domino when the server operates with users. For more information on these Items consult the Lotus Domino documentation. Go to http://www.lotus.com/products/domdoc.nsf, and click the **Lotus Domino Document Manager 3.5** link.

The same names must be used for Entry attribute names when performing Add, Modify, Delete or Lookup operations with the Connector.

- AltFullName
- AltFullNameLanguage
- AltFullNameSort
- Assistant
- AvailableForDirSync
- CalendarDomain
- CellPhoneNumber
- CcMailUserName
- Certificate
- CheckPassword
- Children
- City
- ClientType
- Comment
- CompanyName
- country
- Department
- DocumentAccess
- EmployeeID
- EncryptIncomingMail
- FirstName

- Form
- FullName
- HomeFAXPhoneNumber
- HTTPPassword
- InternetAddress
- JobTitle
- LastName
- Level0
- Level0_1
- Level0_2
- Level0_3
- Level1
- Level1_1
- Level1_2
- Level1_3
- Level2
- Level2_1
- Level2_2
- Level2_3
- Level3
- Level3_1
- Level3_2
- Level3_3
- Level4
- Level4_1
- Level4_2
- Level4_3
- Level5
- Level5_1
- Level5_2
- Level5_3
- Level6
- Level6_1
- Level6_2
- Level6_3
- LocalAdmin
- Location
- MailAddress
- MailDomain
- MailFile
- MailServer
- MailSystem
- Manager
- MessageStorage
- MiddleInitial

- NetUserName
- NoteID
- OfficeCity
- OfficeCountry
- OfficeFAXPhoneNumber
- OfficeNumber
- OfficePhoneNumber
- OfficeState
- OfficeStreetAddress
- OfficeZIP
- Owner
- PasswordChangeDate
- PasswordChangeInterval
- PasswordGracePeriod
- PersonalID
- PhoneNumber
- PhoneNumber_6
- SametimeServer
- ShortName
- Spouse
- State
- StreetAddress
- Suffix
- Title
- Type
- WebSite
- x400Address
- Zip

Domino Server for Unix/Linux

For Domino Users Connector with Domino Server for Unix/Linux, you must update the *ibmditk* and *ibmdisrv* scripts. Add the following two lines in the script, after the PATH definition and before the startup line:

```
LD_LIBRARY_PATH=Domino_binary_folder
export LD_LIBRARY_PATH
```

where *Domino_binary_folder* is the folder containing Domino native libraries, for example, /opt/lotus/notes/latest/sunspa for Solaris, and /opt/lotus/notes/latest/linux for Linux.

Start IBM Tivoli Directory Integrator with the Domino user (do not use **root**). The Domino user is called **notes** unless it is changed during the installation of the Domino Server.

Examples

Go to the *root_directory*/examples/dominoUsersConnector directory of your IBM Tivoli Directory Integrator installation.

See also

"Domino AdminP Connector" on page 73,

"Lotus Notes Connector" on page 77,

Registering Users in Domino,

Registering Users in Domino using Java.

Domino AdminP Connector

The Domino Administration Process is a program that automates many routine administrative tasks. For example, if you delete a user account, the Administration Process locates that user's name in the Domino Directory and removes it, locates and removes the user's name from ACLs, and makes any other necessary deletions for that user. When you put a request in the Domino Administration Requests database the process carries out all required actions.

Overview

The Domino AdminP Connector is a special version of the Lotus Notes Connector. For the Domino AdminP Connector, it has been enhanced to have the capability to sign fields while adding a document to the Domino database. In comparison with the Lotus Notes Connector, the Database parameter is not visible (it is always set to admin4.nsf, the Domino Administration Requests database), and it has a new parameter: **Admin Request Type**.

The Domino AdminP Connector supports the following Connector modes:

- Iterator - iterate over all or a filtered subset of Administration requests
- AddOnly – creates and signs administration requests

Admin requests signing

Domino Administration requests need to be signed before they are added to the admin4.nsf database in order to be further processed by the Administration process. When you sign a document a unique portion of your user ID is attached to the signed note to identify you as author. Otherwise the following error appears:

```
All of the required fields in the request have not been signed.
```

```
Cause of error - An unauthorized person or a non-Domino program edited a posted request.
This indicates a failed security attack.
```

Special coding in the Domino AdminP Connector ensures that all items of the Lotus Domino Document are being signed before the Document is saved.

Note: Even the Lotus Domino administrator should have the rights to "Run Unrestricted methods & operations" in order to be able to sign documents. This can be accomplished using the Domino Administrator by adding that account, for example, administrator/IBM, in the **Server -> Security -> Run Unrestricted methods & operations** list.

Schema

The Domino AdminP Connector has a set of predefined Administration requests schemas. They are described in its configuration file, `tdi.xml`, in a similar manner as the input/output schemas defined in all other Connectors. However, there are two differences:

- the name of the schema is not Input or Output but can be whatever request name is specified,
- the schema should always specify the ProxyAction attribute. It identifies the type of the Domino Administration request. If it is missing, no filtering can be provided for this request type.

Currently, there are only two types of Admin requests bundled in the configuration of this Connector. They have the following definition:

Rename User:

Table 7. Rename User Schema

Attribute	Description
Form	The form of the request. Should be "AdminRequest".
ProxyAction	Corresponds to the id of the request made. For RenameUser the id is "118".

Table 7. Rename User Schema (continued)

Attribute	Description
ProxyAuthor	The author of the request who must have administrative privileges. (for example, CN=administrator/O=IBM).
ProxyNameList	The Domino user's name to be modified.
ProxyNewWebFirstName	The new first name of the user.
ProxyNewWebLastName	The new last name of the user.
ProxyNewWebMI	The new middle name of the user.
ProxyNewWebName	A new UserName to be added.
ProxyProcess	The process of the request. Should be "AdminP".
ProxyServer	The target server. Typically "*".
ProxyWebNameChangeExpires	The expiration period of the request. The default is 21 days.
Type	Type of the request. Should be "AdminRequest".

Rename Group:

Table 8. Rename Group Schema

Attribute	Description
Form	The form of the request. Should be "AdminRequest".
ProxyAction	Corresponds to the id of the request made. For RenameGroup the id is "40".
ProxyAuthor	The author of the request who must have administrative privileges. (for example, CN=administrator/O=IBM).
ProxyNameList	The Domino group's name to be modified.
ProxyNewGroupName	The new name of the group.
ProxyProcess	The process of the request. Should be "AdminP".
ProxyServer	The target server. Typically "*".
ProxyWebNameChangeExpires	The expiration period of the request. The default is 21 days.
Type	Type of the request. Should be "AdminRequest".

These Schemas are returned when you perform a **Discover Attributes** action in the Configuration Editor.

All:

No attributes defined.

The "Rename User" and "Rename Group" schemas define the necessary fields to rename a user or group in a Domino Directory with samples of the values needed. The other schema ("All") is empty and used for any other type of requests; in order to use these you must add new schemas with valid attributes and corresponding new dropdown items.

Configuration

The Connector needs the following parameters:

Session Type

Can be either **IIOP** or **LocalServer**. See “Session types” on page 77.

Domino Server IP Address
The IP hostname or address of the Domino server. You can also specify the IOR:<xxx> string to circumvent automatic discovery of this via HTTP. See the section about the IOR string for more information.

HTTP port
This parameter is used by the Connector to get the IOR string from the Domino HTTP task so as to create an IIOP session.

Username
The username used for IIOP sessions and Local Server sessions.

Password
Internet password for IIOP sessions and Local Server sessions.

Use SSL
Checking this flag causes the Connector to request an encrypted IIOP connection. This flag has meaning when the session type is IIOP only. One of the requirements for using SSL is that the TrustedCerts.class file that is generated every time the DIIOP process starts must be in the classpath. You must copy the TrustedCerts.class to a local path included in the CLASSPATH or have the \Lotus\Domino\Data\Domino\Java of your Domino installation in the classpath.

Admin Request Type
The type of the administration request. The list is retrieved from the configuration file. Available values are "Rename User", "Rename Group" and "All"; the default value is "All".

Support RichText items
Checking this enables Domino RichTextItems to be mapped as such in the Entry. Otherwise they will be converted to plain text.

Attention: As RichTextItems are not serializable, enabling this option will cause an Exception if an attribute is mapped to another Domino database or is used remotely.

Domino Server Name
The name of the server where **Database** is found. Leave blank to use the server you are connecting to (as specified in **Domino Server IP Address**).

Detailed Log
If this parameter is checked, more detailed log messages are generated.

See also
"Domino Users Connector" on page 59,

"Lotus Notes Connector" on page 77

Java API.

Lotus Notes Connector

The Lotus Notes Connector provides access to Lotus Domino databases.

It enables you to do the following tasks:

- retrieve documents and their items from a Notes Database
- create documents
- modify document fields
- delete documents
- perform "lookup" of Notes documents

Note: Lotus Notes Connector requires Lotus Notes to be release 7.0 or higher.

Known limitations

For Lotus Notes Connector using Local Client or Local Server modes only: you might not be able to use the IBM Tivoli Directory Integrator Config Editor to connect to your Notes database. Sometimes, the Notes Connector prompts the user for a password even though the Notes Connector provides it to the Notes APIs. The prompt is written to standard-output, and input from the user is read from standard-input. This prompting is performed by the Notes API and is outside the control of IBM Tivoli Directory Integrator:

- When you run the IBM Tivoli Directory Integrator Server, both standard input and output are connected to the console which enables the user to see the prompt and enter a password. The Notes Connector regains control and continues execution. This means the Connector works as expected.
- When you run the IBM Tivoli Directory Integrator Config Editor, the standard input and output are disconnected from the console so the user cannot see or type anything in response. A connect operation can hang indefinitely waiting for user input.

When the **Session Type** is **LocalClient**, you can start your Notes or Designer client and permit other applications to use its connection by setting a flag in the **File -> Security -> User Security** panel; click Security Basics, and select **Don't prompt for a password from other Lotus Notes-based programs (reduces security)** under "Your Login and Password Settings.". In this case, the Notes Connector (that is, the Notes API) ignores the provided password and reuses the current session established by the Notes or Designer client. The Notes or Designer client must be running to enable IBM Tivoli Directory Integrator to reuse its session.

Note: You can switch to using DIIOP mode to configure your AssemblyLines and switch back to Local Client or Local Server mode when you run the AssemblyLine through IBM Tivoli Directory Integrator Server.

Session types

The following session types are supported (also refer to Supported session types by Connector for more information regarding libraries, setups and incompatibilities with other Domino Connectors):

IIOP This session type uses a TCP connection to the Domino server. The Lotus Notes Connector uses HTTP and IIOP to access the Domino server, so make sure these services are started and accessible from the host where you are running the Lotus Notes Connector.

LocalClient

This session type uses a local installation of Lotus Notes or Designer. The Lotus Notes Connector uses the ID file in use by the local client.

With this session type, the **Username** parameter (dominoLogin) is ignored. The **Password** (dominoPassword) must match the password in the ID file used or the local Notes client prompts for a password. The **Domino Server IP Address**, **IOR String**, **HTTP Port** and Use SSL parameters are disregarded too for this session type.

Note: This can be difficult, for example, when you run an AssemblyLine with standard input or output detached from the console. Always try to run an AssemblyLine in a command line window to detect whether the local client is prompting for the password. Testing shows that the local client ignores the correct **Password** parameter and always prompts for a password. One way of making sure the prompt is avoided is to do the following steps:

1. Start the Notes or Designer client.
2. Go to the **File->Tools->UserID** menu.
3. Check **Don't prompt for a password for other Notes programs**.

LocalServer

Same as for **LocalClient** but uses the local Domino server installation. One difference is that you can specify a valid **Username** and matching **Password**. However, the **Domino Server IP Address**, **IOR String**, **HTTP Port** and Use SSL parameters are disregarded too for this session type.

Connecting with IIOP

The Connector can use IIOP to communicate with a Domino server. To establish an IIOP session with a Domino server, the Connector needs the IOR string that locates the IIOP process on the server.

When you configure the Notes Connector, specify a hostname and, optionally, a port number where the server is located. This *hostname*:*port* string is in reality the address to the Domino server's http service from which the Connector retrieves the IOR string. The IOR string is then used to create the IIOP session with the server's IIOP service (diiop). The need for the http service is only for the discovery of the IOR string. This operation is very simple. The Connector requests a document called **/diiop_ior.txt** from the domino http server that is expected to contain the IOR string. You can replace the *hostname*:*port* specification with this string and bypass the first step and also the dependency of the http server. The diio_ior.txt file is typically located in the `data/domino/html` directory in your Domino server installation directory. Check the Web configuration in the Lotus Administrator for the exact location.

To verify the first step, go to the following URL: http://*hostname*:*port*/diiop_ior.txt where *hostname* is the hostname, and *port* is the port number of your domino server. You receive a document that says IOR: *numbers*. If you get a response similar to this, the first step is verified. If this fails, you must check both the HTTP configuration on the server that it enables anonymous access, and verify that the process is running.

Note: When configuring an IIOP session, in order to be able to browse available databases in the configuration of the Connector in the Config Editor, the Domino server must support that and also allow the various controls be populated with lists of available databases, views, forms and agents. The Domino server setting to make databases available for browsing is located under **Server document -> Internet Protocols -> HTTP tab -> Allow HTTP clients to browse databases**. It must be set to *Yes* and the Domino server must be restarted.

Configuration

The Connector needs the following parameters:

Session Type

Can be one of **IIOP**, **LocalClient** or **LocalServer**. See “Session types” on page 77.

Domino Server IP Address

The IP hostname or address of the Domino server. You can also specify the IOR:<xxx> string to circumvent automatic discovery of this via HTTP. See the section about the IOR string for more information.

HTTP port

This parameter is used by the Connector to get the IOR string from the Domino HTTP task so as to create an IIOP session.

Username
The username used for IIOP sessions and Local Server sessions. Ignored if you use Session type **LocalClient**.

Password
Internet password for IIOP sessions and Local Server sessions. Notes ID file password for Local Client sessions.

Use SSL
Checking this flag causes the Connector to request an encrypted IIOP connection. This flag has meaning when the session type is IIOP only. One of the requirements for using SSL is that the TrustedCerts.class file that is generated every time the DIIOP process starts must be in the classpath. You must copy the TrustedCerts.class to a local path included in the CLASSPATH or have the \Lotus\Domino\Data\Domino\Java of your Domino installation in the classpath.

Support RichText items
Checking this enables Domino RichTextItems to be mapped as such in the Entry. Otherwise they will be converted to plain text.

Attention: As RichTextItems are not serializable, enabling this option will cause an Exception if an attribute is mapped to another Domino database or is used remotely.

Server The name of the server where **Database** is found. Leave blank to use the server you are connecting to (as specified in **Domino Server IP Address**).

Database
The name of the database to use.

Document Selection
The selection used when iterating the data source (i.e., this parameter is only used in **Iterator** mode). You must use valid Lotus Notes select statements. To select entries from the name and address book use the following select statement:

```
Select Form="Person"
```

Always use Formula Search
This flag is used when View is not set and the database accessed is full-text indexed. If you check this flag, the Connector uses Formula statements regardless of whether the database is indexed or not. When a view is specified, full-text searches are always used because View does not support Formula search statements.

Database View
The database view to use.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

UNID Support

UNID is the universally unique ID of a Notes document – it is unique even across database replicas.

The Notes API does not allow the UNID to be used directly in a search filter passed to the Notes/Domino search functions. That is why adding the option of using the UNID in search criteria will be accomplished in the following way:

- If the UNID is present in the Link Criteria all other fields from the criteria will be ignored by the Connector and the Document search will be performed only by UNID.
- If more than one UNID is specified, the first met will be considered.
- If the match operation is not equals, an exception is thrown.
- If no document is found with the specified UNID, an exception is thrown.
- If a document with the specified UNID is found, the Connector checks whether it is not deletion stub. If it is not a deletion stub, the document is processed.

- If there is no UNID in the Link Criteria – the search filter will be built as in TDI 6.0.

The described functionality will not cause backward compatibility issues because the Notes API does not allow UNIDs in formulas/search filters.

Support of RichText attributes

Lotus Domino documents contain items of type `lotus.domino.RichTextItem`. Traditionally, when read, such items are received in the attribute map of an Entry as plain text and vice-versa – written as plain text back to the domino document. The Connector supports additional functionality for these kind of items:

Iterator mode

By checking the parameter **Support RichText items** you modify the Connector's behavior such that when the Connector reads documents from a Domino database, if a lotus.domino.RichTextItem is found it is put in the Entry as a lotus.domino.RichTextItem, otherwise it is put in the Entry as String.

Add and Update mode

When adding/updating a Lotus Domino document there is an option to provide a lotus.domino.RichTextItem object in the Entry, to be written to that document. This RichTextItem could be received from elsewhere in the AssemblyLine, or created by a script.

Example scripts: Creating a RichTextItem:

```
var rti = NotesConnectorName.connector.getDominoSession().getDatabase("", <database_name>).createDocument().createRichTextItem("Body");
var header = NotesConnectorName.connector.getDominoSession().createRichTextStyle();

header.setBold(lotus.domino.RichTextStyle.YES);
header.setColor(lotus.domino.RichTextStyle.COLOR_YELLOW);
header.setEffects(lotus.domino.RichTextStyle.EFFECTS_SHADOW);
header.setFont(lotus.domino.RichTextStyle.FONT_ROMAN);
header.setFontSize(14);

rti.appendStyle(header);
rti.appendText("Sample text which will be formatted with the above style.");

work.setAttribute("Body", rti);
```

Extracting attachment from a RichTextItem:

```
var doc = NotesConnectorName.connector.getDominoSession().getDatabase("", <database_name>).getAllDocuments().getFirstDocument();
      if (doc.hasEmbedded()) {
        var body = doc.getFirstItem("Body");
        var rtnav = body.createNavigator();
        if (rtnav.findFirstElement(
        lotus.domino.RichTextItem.RTELEM_TYPE_FILEATTACHMENT)) {
          do {
            var att = rtnav.getElement();
            var path = "c:\\Files\\" + att.getSource();
            att.extractFile(path);
            main.logmsg(path + " extracted");
          } while (rtnav.findNextElement());
        }
        else
          main.logmsg ("No attachments");
      }
      else
        main.logmsg ("No attachments or embedded objects");
```

For more information and examples see the Domino documentation at: http://www-128.ibm.com/developerworks/lotus/documentation/dominodesigner/

RichText limitations: The `lotus.domino.RichTextItem` class is not serializable and items of this type can not be transferred through RMI. This will cause a java.io.NotSerializableException when an Object of this type is accessed through Remote Server APIs. Once a not serializable object gets into the Entry the whole Entry becomes not serializable.

Note: This also applies to the `lotus.domino.DateTime` class.
Also, this serialization limitation restricts lotus.domino.RichTextItems to be transferred between different

Domino databases. For this reason a RichTextItem can be added/updated only with another RichTextItem from the *same database* or with a RichTextItem created with a script using the domino API, but still for the same database.

Setting quota and file ownership

The Connector can only directly manipulate Lotus Notes database entries, but not database properties. However, database quotas can be set by means of scripting, using a Script Component and a configured Lotus Notes Connector. The sample code below should set file size quota and write the desired MailOwnerAccess to the ACL.

```
//NotesIterator is the NotesConnector name in the AssemblyLine
var db = NotesIterator.connector.getDominoDatabase(null);
//uses the public getDominoDatabase(...) method of the NotesConnector class;
//giving null for method parameter will return the database configured in the Connector
main.logmsg("Old quota: " + db.getSizeQuota());
//should print the old database size quota
db.setSizeQuota(5000);
//sets the size quota to 5000KB
main.logmsg("New quota: " + db.getSizeQuota());
//will print the new database size quota in kilobytes, i.e. 5000
var acl = db.getACL();
//get the database access control list
var ACLEntry = acl.createACLEntry("DesiredNotesUser", lotus.domino.ACL.LEVEL_MANAGER);
//create new ACL Entry
ACLEntry.setUserType(lotus.domino.ACLEntry.TYPE_PERSON); //set user type equal to Person
acl.save();
//save the access control list
```

Security

To have IBM Tivoli Directory Integrator access your Domino server, you must enable it through **Domino Administrator -> Security -> IIOP restriction**. The user account you configured for the IBM Tivoli Directory Integrator to use must belong to a group listed under **Run restricted Java/Javascript** and **Run unrestricted Java/Javascript**.

The Domino Web server must be configured to enable anonymous access. If not, the current version of the Notes Connector cannot connect to the Domino IIOP server.

Note: If you want to encrypt the **HTTPPassword** field of a Notes Address Book, add the following code to your AssemblyLine:

```
var pwd = "Mypassword";
var v = dom.connector.getDominoSession().evaluate
("@Password(\"" + pwd + "\")" ) ;
ret.value = v.elementAt(0);
```

This code uses Domino's password encryption routines to encrypt the variable *pwd*. It can be used anywhere that you want to encrypt a string using the **@Password** function that Domino provides. A good place to use this code is in the **Output Map** for the **HTTPPassword** attribute.

See also

Wikipedia on Lotus Notes.

ITIM DSMLv2 Connector

This Connector is used in solutions which require communication with IBM Tivoli Identity Manager (ITIM).

The ITIM server provides a communication interface which uses a ITIM-proprietary version of DSMLv2. This ITIM-proprietary version of DSMLv2 doesn't fully comply with the DSMLv2 specification. Hence the Connector name – *ITIM DSMLv2 Connector*.

This Connector is used for both:

- retrieving provisioning data, and
- feeding provisioning data

using the ITIM-proprietary DSMLv2 communication interface.

The version of ITIM supported is 5.0 and higher.

The Directory Services Markup Language v1.0 (DSMLv1) enables the representation of directory structural information as an XML document. DSMLv2 goes further, providing a method for expressing directory queries and updates (and the results of these operations) as XML documents.

Note: This Connector is *specially designed* for use with ITIM; for generic use, use the DSMLv2Soap Connector and/or the DSMLv2SoapServer Connector instead.

The ITIM DSMLv2 Connector which connects to a IBM Tivoli Identity Manager Server repository using DSML over HTTP.

The Connector connects to the DSMLv2 ITIM event handler (introduced in ITIM 4.5) that allows the import of data into ITIM with ITIM acting as a DSMLv2 server. Therefore, only ITIM Server 4.5 and above is supported. The ITIM DSMLv2 Connector uses the ITIM DSML JNDI driver "dsml2.jar", to connect to and interact with the ITIM Server. Deployment of the DSMLv2 Connector uses JNDI queries to interact with the ITIM repository.

The Connector supports the **AddOnly**, **Delete**, **Iterator**, **Lookup** and **Update** modes.

Note: This component is not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.

Skip Lookup in Update and Delete mode

The ITIM DSMLv2 Connector supports the **Skip Lookup** general option in Update or Delete mode. When it is selected, no search is performed prior to actual update and delete operations. It requires a **Name** parameter (for example, $dn for LDAP) to be specified in order to operate properly.

Using the Connector with ITIM Server

When connecting to a ITIM Server the following URL should be specified in the ITIM DSMLv2 Connector: http://<*ITIM_Server_host:ITIM_Server_port*>/enrole/dsml2_event_handler; for example, "http://192.168.113.12:9080/enrole/dsml2_event_handler".

The following limitations apply to ITIM DSMLv2 Connector modes when interacting with ITIM Server:

- Iterator mode – will only work if the JNDI filter specified matches exactly one Entry; if the filter matches more than one Entry, no Entries will be returned.
- Lookup, Update and Delete – will only work correctly if the link criteria specified result in finding exactly one Entry; if the link criteria match more than one Entry, the Connector will act as if the link criteria matched no Entries.

When interacting with ITIM Server, all JNDI queries and filters, used either from the GUI or in scripting (in Advance Search Criteria, for example) must be enclosed in brackets, for example "(uid=user1)".

HTTPS (SSL) Support

In order to use a secure HTTPS connection to the DSMLv2 Server, the provider URL specified must begin with "https://" and the server's certificate must be included in Tivoli Directory Integrator's trust store.

Configuration

The ITIM DSMLv2 Connector needs the following parameters:

Provider URL
The URL for the connection.

Referrals
Specifies how referrals encountered by the LDAP server are to be processed. The possible values are:

- **follow** - Follow referrals automatically.
- **ignore** - Ignore referrals
- **throw** - Throw a ReferralException when a referral is encountered. You need to handle this in an error Hook.

Authentication Method
The authentication method.

Login username
The principal name (for example, username).

Login password
The credentials (for example, password).

Name parameter
Specify which parameter in the AssemblyLine entry is used for naming the entry. This is used during add, modify and delete operations and returned during read or search operations. If not specified, "$dn" is used.

Provider Params
A list of extra provider parameters you want to pass to the provider. Specify each "parameter:value" on a separate line.

Search Base
The search base to be used when iterating the directory. Specify a distinguished name. Some directories allow you to specify a blank string which defaults to whatever the server is configured to do. Other directory services require this to be a valid distinguished name in the directory.

Search Filter
The search filter to be used when iterating the directory.

Search Scope
The search scope to be used when iterating the data source. Possible values are:

- subtree - search all levels from search base and below
- onelevel - search only immediate children of search base

Detailed Log
If this parameter is checked, more detailed log messages are generated.

See also

"ITIM Agent Connector" on page 129,

DSML Identity Feed.

DSMLv2 SOAP Connector

The DSMLv2 SOAP Connector implements the DSMLv2 standard. The Connector is able to:

- Execute DSMLv2 requests against a DSML Server.
- Provide the option to use DSML SOAP binding.
- Internally instantiate, configure and use the HTTP Parser to create HTTP requests and parse HTTP responses.
- Internally instantiate, configure and use the DSMLv2 Parser to create DSMLv2 request messages and parse DSMLv2 response messages.

Supported Connector Modes

The Connector mode determines the type of DSML operation the Connector requests. The DSMLv2 SOAP Connector supports the following modes:

AddOnly
The DSMLv2 SOAP Connector sends DSMLv2 addRequest and receives a DSMLv2 addResponse message.

Iterator
The DSMLv2 SOAP Connector sends a DSMLv2 searchRequest operation with a Search Base, Search Filter and Search Scope taken from the current Connector configuration. The DSML server returns a DSMLv2 searchResponse message with multiple searchResultEntry elements. The Connector cycles through the DSML searchResultEntry elements and delivers each one in a separate AssemblyLine iteration.

Lookup
The DSMLv2 SOAP Connector sends a DSMLv2 searchRequest with a Search Filter constructed from the Connector's Link Criteria. The DSML server returns a DSMLv2 searchResponse message that is returned as the Entry found. If there are multiple searchResultEntry elements in the searchResponse message, you must process them in an `On Multiple Entries` hook.

Delete The Connector creates and sends a DSML deleteRequest as per the Link Criteria. The DSML server returns a deleteResponse message.

Update
If the `$dn` Attribute in the work Entry is equal to the `$dn` Attribute of the Entry to be updated, the Connector sends a modifyRequest DSMLv2 request and receives a modifyResponse response; otherwise a modDnRequest request is sent to the DSML server and a modDnResponse response is received.

Delta In Delta mode, it is the AssemblyLine that, depending on the Entry tagging, decides which Connector method to invoke and what DSMLv2 request will be sent. Delta tagging at the Attribute level is handled by the DSMLv2 Parser and delta information is incorporated into the resulting DSMLv2 request.

The DSMLv2 SOAP Connector detects in its modEntry method if the "newrdn" attribute exists and if yes it replaces the rdn in the target $dn with the new value. Then a modDnRequest request is sent to the DSML server and a modDnResponse response is received.

CallReply
In CallReply mode, the Connector provides the work Entry to the DSMLv2 Parser and sends the DSMLv2 message produced by the DSMLv2 Parser. The response from the DSMLv2 Server is passed directly to the DSMLv2 Parser, and the Entry produced is returned by the Connector. You must assign the correct request type, because the Connector will not automatically set any DSMLv2 element. In particular, the CallReply mode can be used to send DSMLv2 extended operations. See “Extended Operations” on page 86 for more information.

Extended Operations

In CallReply mode, the DSMLv2 SOAP Connector can send DSMLv2 extended operations. Extended operations are identified by their Operation Identifier (OIDs). For example, the OID of the extended operation for retrieving a part of the log file of the IBM Tivoli Directory Server is `1.3.18.0.2.12.22`.

Extended operations can also have a value property, which is a data structure containing input data for the corresponding operation. The `value` property of the extended operation must be Basic Encoding Rules (BER) encoded and then base-64 encoded in the DSMLv2 message. The user of the DSMLv2 SOAP Connector is responsible only for BER encoding the value property. The Connector will automatically base-64 encode the data when creating the DSMLv2 message.

Two classes are used for BER encoding and decoding: BEREncoder and BERDecoder, located in the`com.ibm.asn1` package.

The following example illustrates sending a DSMLv2 extended operation request and the processing of the response:

1. Place the following script code in Output Map for attribute `dsml.extended.requestvalue`:

```
enc = new Packages.com.ibm.asn1.BEREncoder();
serverFile = 1;  //slapdErrors log file

nFirstLine = new java.lang.Integer(7200);
nLastLine = new java.lang.Integer(7220);

seq_nr = enc.encodeSequence();
enc.encodeEnumeration(serverFile);

enc.encodeInteger(nFirstLine);
enc.encodeInteger(nLastLine);

enc.endOf(seq_nr);
var myByte = enc.toByteArray();

ret.value = myByte;
```

2. Place the following script code in the `After CallReply` hook of the Connector:

```
var ba = conn.getAttribute("dsml.response").getValue(0);
bd = new Packages.com.ibm.asn1.BERDecoder(ba);

main.logmsg("SLAPD log file:");
main.logmsg(new java.lang.String(bd.decodeOctetString()));
```

SOAPAction Header

The DSMLv2 SOAP Connector by default always sends an empty header for the SOAPAction header. The OASIS Standard around SOAP states the this: "Each SOAP request body contains a single batchRequest. A SOAP node SHOULD indicate in the 'SOAPAction' header field the element name of the top-level element in the <body> of the SOAP request." It is valid for this header to be empty but it should optionally be something that can be set. Additionally, some vendors have defined the header to be mandatory in their DSML definitions (Sun is an example, see http://docs.sun.com/source/816-6700-10/DSML.html).

If needed, you can set the SOAPAction Header yourself by means of the **SOAPAction Header** parameter.

Configuration

The DSMLv2 SOAP Connector uses the following parameters:

DSMLv2 Server URL
: Specifies the URL of the DSMLv2 Server.

Authentication Method
Specifies the type of HTTP authentication. If the type of HTTP authentication is set to `Anonymous`, then no authentication is performed. If `HTTP basic authentication` is specified, HTTP basic authentication is used with user name and password as specified by the `username` and `password` parameters.

Username
The user name used for HTTP basic authentication.

Password
The password used for HTTP basic authentication.

Binary Attributes
Specifies a comma-delimited list of attributes that will be treated by the Connector as binary attributes. This parameter has the following default list of attributes that you can change:

- photo
- personalSignature
- audio
- jpegPhoto
- javaSerializedData
- thumbnailPhoto
- thumbnailLogo
- userPassword
- userCertificate
- authorityRevocationList
- certificateRevocationList
- crossCertificatePair
- x500UniqueIdentifier
- objectGUID
- objectSid

Search Base
Specifies the starting point for searches when iterating.

Search Filter
Specifies the LDAP filter used when iterating.

Search Scope
The search scope to be used when iterating. Possible values are:

- subtree
- onelevel

The default is subtree.

Soap Binding
When this parameter is enabled, the Connector sends and receives SOAP DSML messages. Otherwise, the DSML messages are not wrapped in SOAP.

SOAPAction Header
The SOAPAction header value to include when SOAP binding is enabled. The default header value is empty.

Detailed Log
Turns on debug messages. This parameter is common to all Tivoli Directory Integrator components.

DSMLv2 SOAP Server Connector

The DSMLv2 SOAP Server Connector listens for DSMLv2 requests over HTTP. Once it receives the request, the Connector parses the request and sends the parsed request to the AssemblyLine workflow for processing. The result is sent back to the client over HTTP.

The DSMLv2 SOAP Server Connector is able to:

- Execute DSMLv2 requests against a DSML Server.
- Provide the option to use DSML SOAP binding.
- Internally instantiate, configure and use the HTTP Parser to create HTTP requests and parse HTTP responses.
- Internally instantiate, configure and use the DSMLv2 Parser to create DSMLv2 request messages and parse DSMLv2 response messages.
- Process each event in a separate thread, allowing the Connector to process several DSMLv2 events in parallel.

Extended operations

The DSMLv2 SOAP Server Connector supports extended operations. The `value` property of the extended operation is automatically base-64 decoded from the DSMLv2 message. You must then properly Basic Encoding Rules (BER) decode this value. You must also BER encode the `responseValue` property represented by the `dsml.response` Entry Attribute. The Connector will automatically base-64 encode the data when creating and sending the DSMLv2 response.

You can use the following two helper classes to BER encode and decode data:

- com.ibm.asn1.BEREncoder
- com.ibm.asn1.BERDecoder

Note: The schema of the extended operations cannot be automatically determined by the Connector. There is no metadata that describes the structure of an extended operation request.

The following example illustrates an extended operation request to return a part of the IBM Tivoli Directory Server log:

```
var name = work.getString("dsml.extended.requestname");
   var ba = work.getAttribute("dsml.extended.requestvalue").getValue(0);

decoder = new Packages.com.ibm.asn1.BERDecoder(ba);
iSecuence = decoder.decodeSequence();
fileNumber = decoder.decodeEnumeration();
firstLine = decoder.decodeIntegerAsInt();
lastLine = decoder.decodeIntegerAsInt();

main.logmsg("Operation: " + name);
main.logmsg("File: " + fileNumber);
main.logmsg("First line: " + firstLine);
main.logmsg("Last line: " + lastLine);

// send the response, assuming this sample string is the log file content
var str = new java.lang.String("Apr 13 16:18:18 2005  Entry cn=chavdar kovachev,o=ibm,c=us already exists.");

enc = new Packages.com.ibm.asn1.BEREncoder();
enc.encodeOctetString(str.getBytes());
myByte = enc.toByteArray();

work.setAttribute("dsml.response", myByte);
work.setAttribute("dsml.responseName", "1.3.18.0.2.12.23");
work.setAttribute("dsml.resultdescr", "success");
```

Configuration

The DSMLv2 SOAP Server Connector uses the following parameters:

Dsml Port
Specifies the TCP port on which the DSMLv2 SOAP Server Connector is listening.

Connection Backlog
Specifies the maximum queue length for incoming connections. If a connection request arrives when the queue is full, the connection will be refused.

HTTP Basic Authentication
Determines if clients must provide HTTP basic authentication.

Auth Realm
Specifies the authentication realm sent to the client when requesting HTTP Basic authentication

Binary Attributes
Specifies a comma-delimited list of attributes that will be treated by the Connector as binary attributes.

This parameter has the following default list of attributes that you can change:

- photo
- personalSignature
- audio
- jpegPhoto
- javaSerializedData
- thumbnailPhoto
- thumbnailLogo
- userPassword
- userCertificate
- authorityRevocationList
- certificateRevocationList
- crossCertificatePair
- x500UniqueIdentifier
- objectGUID
- objectSid

Use SSL
If checked, Secure Sockets Layer (SSL) will be used while initializing the connector.

Require Client Authentication
If checked, the connector will require client authentication using SSL.

Chunked Transfer Encoding
If checked, the HTTP body of the response message is transferred as a series of chunks.

Soap Binding
If checked, the Connector sends and receives SOAP DSML messages. Otherwise, the DSML messages are not wrapped in SOAP.

Detailed Log
Turns on debug messages. This parameter is common to all Tivoli Directory Integrator components.

EIF Connector

IBM Tivoli Directory Integrator leverages the capabilities of the Event Integration Facility in the process of integration with enterprise systems like Netcool/OMNIbus and IBM Tivoli Enterprise Console. EIF enables Tivoli Directory Integrator to create and send alerts and status information that can be recognized by the Netcool/OMNIbus Event Management system as events.

The EIF Connector allows Tivoli Directory Integrator to both send and receive EIF event messages, facilitating bi-directional communications with EIF-capable systems like TEC and Netcool/Omnibus.

Sending is done using the Connector AddOnly mode, while reading events is handled by Iterator mode.

Introduction to IBM Tivoli Netcool/OMNIbus

IBM Tivoli Netcool®/OMNIbus is a service level management (SLM) system that collects enterprise-wide event information from many different network data sources and presents a simplified view of this information to operators and administrators.

This information can then be:

- Assigned to operators.
- Passed to helpdesk systems.
- Logged in a database.
- Replicated on a remote Tivoli Netcool/OMNIbus system.
- Used to trigger automatic responses to certain alerts.

Tivoli Netcool/OMNIbus can also consolidate information from different domain-limited network management platforms in remote locations. By working in conjunction with existing management systems and applications, Tivoli Netcool/OMNIbus minimizes deployment time and enables employees to use their existing network management skills.

Tivoli Netcool/OMNIbus tracks alert information in a high-performance, in-memory database and presents information of interest to specific users through individually configurable filters and views. Tivoli Netcool/OMNIbus automation functions can perform intelligent processing on managed alerts.

The IBM Tivoli Netcool/OMNIbus documentation Web site is available at http://publib.boulder.ibm.com/infocenter/tivihelp/v8r1/index.jsp?toc=/com.ibm.netcool_OMNIbus.doc/toc.xml

Introduction to Tivoli Enterprise Console

The IBM Tivoli Enterprise Console (TEC) product is a rule-based event management application that integrates system, network, database, and application management to help ensure the optimal availability of an organization's IT services.

The Tivoli Enterprise Console product:

- Provides a centralized, global view of your computing enterprise.
- Collects, processes, and automatically responds to common management events, such as a database server that is not responding, a lost network connection, or a successfully completed batch processing job.
- Acts as a central collection point for alarms and events from a variety of sources, including those from other Tivoli software applications, Tivoli partner applications, custom applications, network management platforms, and relational database systems.

The Tivoli Enterprise Console product helps you effectively process the high volume of events in an IT environment by:

- Prioritizing events by their level of importance.

- Filtering redundant or low-priority events.
- Correlating events with other events from different sources.
- Determining who must view and process specific events.
- Initiating automatic corrective actions, when appropriate, such as escalation, notification, and the opening of trouble tickets.
- Identifying hosts and automatically grouping events from the hosts that are in maintenance mode in a predefined event group.

Refer to the *IBM Tivoli Enterprise Console User's Guide Version 3.9*, SC32-1235, for more information about this product and its components.

Introduction to the Event Integration Facility

The IBM Tivoli Event Integration Facility (EIF) is an event distribution and integration point for the event console. With the Tivoli Event Integration Facility toolkit, you can build event adapters and integrate them into the IBM Tivoli Enterprise Console environment.

IBM Tivoli Enterprise Console adapters are the integration link. Adapters collect events, perform local filtering, translate relevant events to the proper format for the event console, and forward these events to the event server. A variety of adapters for systems, Tivoli software applications, and third-party applications are available. To monitor a source (such as a third-party or custom application) that is not supported by an existing adapter, you can use Tivoli Event Integration Facility to create an adapter for the source.

You can use Tivoli Event Integration Facility to:

- Specify the event information to send to the event server for processing.
- Create an adapter to filter, translate, and then forward event information to the event server.
- Filter and correlate events near the source by using state correlation.
- Create an application that can receive events.

Refer to the *IBM Tivoli Event Integration Facility User's Guide Version 3.8*, GC32-0691-01, for more information.

Schema

The EIF Connector schema will be retrieved from the Netcool gateway mapping file if it is specified in the **Schema File** (eifSchemaFile) configuration parameter. For more information, see "Configuration" on page 93.

Iterator mode

When the EIF Connector is in Iterator mode, it will feed the AssemblyLine with entries that comply with the following structure:

Table 9.

Attribute name	Value
className	String
slotname	String
slotname	String
...	

where slotname is the name of the slot specified in the received event.

AddOnly mode

When the EIF Connector is in AddOnly mode, it will send an event to a remote system. The event to be sent is specified as an Entry. The connector expects the provided to it entry to comply with the following structure:

Table 10.

Attribute name	Value
className	String
slotname	String
slotname	String
...	

Configuration

The EIF Connector expects the following parameters:

EIF Config File
: This text field contains the path to the config file or a block of text representing properties recognized by the underlying library. When properties are being specified, an "!" should be the first character in this field.

Schema File
: This optional string parameter contains the path to the Netcool EIF gateway's mapping file used for retrieving the schema. If the specified file can not be found, opened or read an error message is logged and only the "msg"attribute is added to the schema (old behavior).

Break on Error
: This Boolean parameter specifies whether the connector should throw an error if unable to establish connection initially. The default value is *true*.

GetNext Timeout
: This numeric parameter specifies the time to wait (in seconds) for an event message to be delivered (-1 - wait forever, 0 - get the next available message without waiting, N - number of seconds to wait). The default value is -1.

Terminate Timeout
: This numeric parameter specifies the time to wait (in seconds) when closing the connection with the remote server. The default value is 120.

Detailed Log
: Check this parameter for more detailed messages in the log.

File system Connector

The file system Connector is a transport Connector that requires a Parser to operate. The file system Connector reads and writes files available on the system it runs on. Concurrent usage of a file can be controlled by means of a locking mechanism.

Note: This Connector can only be used in Iterator or AddOnly mode, or for the equivalent operations in Passive state.

Configuration

The Connector needs the following parameters:

File Path
The name of the file to read or write.

Timeout (in seconds)
When this parameter is specified as a positive number, the Connector waits for available data when reading from a file (that is, the Connector is in Iterator mode). Specify **0** (zero) to wait forever, or any other number which specifies the number of seconds to wait before signalling end of file. Setting this parameter to **0** (zero) causes the Connector to simulate the UNIX-style **tail -f** command.

If you have requested a lock on the file (using the **Lock File** parameter), the Timeout parameter instead specifies how long to wait to acquire the lock. An unspecified or negative number means "wait forever".

Append on Output
If set, the Connector appends instead of overwriting when the file is opened for writing.

Lock file
When writing, acquire an exclusive lock on the file being written. When reading, acquire a shared lock.

The lock is acquired when the Connector is initialized, and released when the Connector is closed.

If one Connector (A) has acquired an exclusive lock on a file, and another Connector (B) tries to open it, then Connector B will either wait for he lock to be released, or an error will be thrown. If Connector B has checked the exclusive Lock parameter, it will wait; if Connector B has not checked the exclusive Lock parameter, an error will be thrown.

The locking mechanism is Operating System dependent. Note that file locking can cause deadlocks, especially if more than one file is locked per AssemblyLine.

For more information, see http://java.sun.com/j2se/1.5.0/docs/api/java/nio/channels/FileChannel.html#lock()

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Parser In the Parser tab, you can configure the name of a Parser to access the contents of the file by selecting a Parser in the "Inherit from:" button.

See also

"URL Connector" on page 285.

Form Entry Connector

This connector feeds an AssemblyLine with entries provided as the connector's parameter. It works like a regular connector, but without having a separate input file. Conceptually, this connector is useful for feeding a test AssemblyLine with test cases which are actually stored as part of the config file. Also this component may come in quite handy when you need to parse data inside the AssemblyLine resulting in a series of entries that you want to iterate over. In this case, you can attach the Form Entry Connector to a Connector Loop and then map the byte stream to the Raw Data Text parameter.

This connector supports Iterator Mode only.

Using the Connector

This connector feeds an AssemblyLine with raw data provided as a parameter to the connector. The connector uses the configured parser to parse the raw data, creates a valid entry and passes it to the underlying AssemblyLine.

Configuration

The Form Entry Connector has two parameters:

Infinite Loop
: This parameter enables looping trough the input data. When enabled, the connector cycles trough the input raw data indefinitely. This can be useful when stress testing AssemblyLine components.

Raw Data Text
: These are the input entries, saved in UTF-8 format. This parameter can be set at runtime, by using the `setParam()` method. The default value of this parameter is:

```
first:John
last:Smith
.
id:2
first:Jane
last:Doe
```

This text can be easily parsed with the Simple Parser and a resulting entries dump would look like this:

```
CTGDIS003I *** Start dumping Entry
Operation: generic
Entry attributes:
 last (replace): 'Smith'
 first (replace): 'John'
 id (replace): '1'
CTGDIS004I *** Finished dumping Entry

CTGDIS003I *** Start dumping Entry
Operation: generic
Entry attributes:
 last (replace): 'Doe'
 first (replace): 'Jane'
 id (replace): '2'
CTGDIS004I *** Finished dumping Entry
```

FTP Client Connector

The FTP Client Connector is a transport Connector that requires a Parser to operate. The Connector reads or writes a data stream that can either be a file or a directory listing. Think of the FTP Client Connector as a remote read/write facility, not something you use to transfer files.

This Connector supports FTP Passive Mode, as per RFC959. Passive Mode reverses who initiates the data connection in a file transfer. Normally the server initiates a data connection to the client (after a command from the client), whereas passive mode enables the client to initiate the data connection. This makes it easier to transfer files when the client is behind a firewall.

Notes:

1. Iterator mode supports the operations **get** and **list**; AddOnly supports **put**.
2. This Connector is not intended for transferring binary files.

With proper configuration, this Connector supports FTP over SSL (FTPS) connections, to provide secure transfers.

SSL support

The FTP Client Connector supports FTPS and can perform secure transfers. This involves the use of a SSL/TLS layer below the standard FTP protocol to encrypt the control and/or data channels used by FTP. There are two common uses of FTPS:

- *Implicit FTPS* is a widely implemented style in which the client connects to a different control port (from the default 21), and an SSL handshake is performed before any FTP commands are sent. The entire FTPS session is encrypted. Implicit FTPS does not allow for negotiation and the client should immediately challenge the FTPS Server with the TLS/SSL handshake. If the control channel is unencrypted, any subsequent data channels must also be unencrypted (no SSL); if the control channel is encrypted, the subsequent data channels may be clear or encrypted. The Internet Assigned Numbers Authority (IANA) officially designates port 990 as the FTPS control channel port and port 989 as the FTPS data channel port.
- *Explicit FTPS* or *FTPES*. According to this method, the client connects using clear text on port 21 and may negotiate a secure TLS connection during the FTP setup or at any time thereafter. The server may allow non-encrypted FTP in case negotiation fails. Encrypted data channels and encryption on the control channel can be set up and torn down by the client at any time.

The FTP Client Connector supports only implicit FTPS, so an SSL handshake must be performed before any transfer. As stated above the FTP protocol uses two channels to operate. The control (command) channel is used for sending commands to the FTP server and the data channel for data transfer. In order to allow greater granularity, the FTP Client Connector allows you to turn on SSL support for each of the channels.

Using the **Security** parameter, you can specify the following options: **None**, **Use SSL on control channel**, **Use SSL on control and data channels**. The first implies that no SSL support will be provided and no security benefits can be expected.

When **Use SSL on control channel** is selected, the control (command) channel uses SSL. In this case the certificate used by the FTP server must be added to the truststore of IBM Tivoli Directory Integrator (this truststore is set by the `javax.net.ssl.trustStore` property in the `solution.properties` file). That way the client can authenticate the server and communication will succeed. Also when using this option, remember to change the port used by the connector to the one that the server uses for FTP/SSL connections (the default is 990).

The other option providing SSL support is **Use SSL on control and data channels**. When this is selected, the client will attempt to negotiate a secure data channel besides securing the control channel. This is done by sending "PBSZ 0" and "PROT P" commands to the server. The PBSZ command defines the largest

buffer size to be used for application-level encoded data sent or received on the data connection. However, since TLS/SSL handles blocking of data, a '0' parameter is used. The other command (PROT) defines the protection used for FTP data connections, where the "P" parameter stands for Private - TLS/SSL will be used, which provides Integrity and Confidentiality protection.

The **Security** parameter lists the allowed set of security options for the FTP Client Connector. However, when the connector is created using scripts there is one other option. Since its security parameters are passed as arguments when it connects to the FTP server (for example, `connect(String host, String user, String password, boolean useSSLonCommandChannel, boolean useSSLonDataChannel)`), it is possible to enable SSL on the data channel and not on the control channel. This configuration implies that the client must connect to the SSL/TLS port of the server sending a plaintext message. The attempt certainly won't succeed, so the FTP Client Connector checks for this case and an error message is displayed when the AssemblyLine is started.

As stated above, the FTP Client Connector can operate in two modes: *Active* and *Passive*. In Passive mode, the FTP server waits for connections from the FTP Client Connector (for the command and data channels). When this occurs the server sends its certificate to the client and SSL communication is possible. In Active mode the situation is the same for the command channel, but this time the client listens for connections (for the data channel). In normal cases this would require the client to send its certificate to the server for validation. To overcome this problem, the SSL session is run in client mode – this means that the SSL roles are reversed (the TCP server acts as client and the TCP client as server, so again the server will send its certificate to the client). This is achieved by the `setUseClientMode(true)` method.

Character Encoding

The FTP Client Connector uses a configured Parser for reading and writing. Therefore data is read from/written to the FTP server using this parser's **Character Encoding** parameter. If no such parameter is specified, the default character encoding of the platform running the IBM Tivoli Directory Integrator is used.

Configuration

The Connector needs the following parameters:

FTP Hostname
The hostname or IP address on which the FTP Server resides that the Connector will connect to.

FTP Port
The FTP TCP port (defaults to **21**).

Login User
The login username.

Login Password
The login password.

Operation
The intended operation. Select **get** to read a file (Iterator), **put** to write a file (Add Only), or **list** to do a directory listing (Iterator).

Remote Path
Initial remote directory (for list) or file (for get/put) to access.

Transfer Mode
ASCII or Binary. ASCII is the only supported mode.

Passive Mode
When this checkbox is enabled, specifies that the FTP Client Connector will connect to the FTP Server in passive mode instead of active mode. This parameter is ignored on an IPv6 connection, since IPv6 *always* uses passive mode.

Security
Depending on the option selected, the FTP Client Connector: won't use a SSL secure connection; will use one for the control channel, or will use one for both the control and data channels. Available values are:

- None
- SSL_control_channel - use SSL on control channel
- SSL_control_data_channels - use SSL on control and data channels

Detailed Log
If this parameter is checked, more detailed log messages are generated.

From the Parser pane, you select the mandatory Parser. For example, Line Reader is a useful parser for list, or if you simply want to copy one file. The select dialog is activated by pressing the top-left **Select Parser** button.

See also

"The FTP object" on page 526,
"URL Connector" on page 285,
"Old HTTP Client Connector" on page 111,
"Old HTTP Server Connector" on page 121.

GLA Connector

Introduction

The GLA Connector processes log files and transforms them to Common Base Event (CBE) objects, which are then fed into the AssemblyLine.

It uses Generic Log Adapter (GLA) technology, part of IBM's Autonomic Computing Toolkit, to process log files and transform their contents into Common Base Event format. The Autonomic Computing Toolkit website contains (in addition to general information and documentation) a variety of downloadable software, including the Eclipse plug-in to edit GLA configuration files.

Adapter configuration file

An adapter configuration file, prepared externally using the Adapter Configuration Editor Eclipse plug-in is used in conjunction with the GLA Connector. It provides the tooling to create the specific parser rules that are used by the GLA Connector at runtime to create Common Base Event objects, that is, the configuration file contains information about the log file which will be processed. From this file all the CBE objects will be later created. The logic for parsing the objects is also implemented in the adapter configuration file. In the Eclipse GLA plug-in there are several examples of such configuration files, made to process some well known application log files. You can also create your own configuration files using the Eclipse's user interface.

In both cases, either using an already created configuration file or creating a new configuration file, you should note that a specially made outputter called *TDIOutputter* needs to be configured. This should be done because when the CBE objects are created, the TDIOutputter sends these CBE objects to the GLA Connector (which can then send them into the AssemblyLine using the ordinary mapping mechanism).

Using more than one outputter in the configuration file

It is not possible to use more than one TDIOutputter. If two or more TDIOutputters are configured in the adapter configuration file an Exception will be thrown when the GLA Connector tries to get the correlation ID from the TDIOutputter. When there is more than one TDIOutputter configured, the GLA Connector which uses the configuration file does not know which of the multiple TDIOutputters to use—it is not possible to get the CBE Objects from the correct TDIOutputter.

However, it is possible to define more than one outputter as long as it is not a TDIOutputter. For example, you could combine the TDIOutputter with a FileOutputter. This will cause all CBE objects to be sent (and saved) both to a file and to the GLA Connector.

Configuration

To configure the GLA Connector you must have a valid adapter configuration file. The path to the file must be set in the Connector **Config File Path** parameter. The configuration file is being checked for validity and if this is not a valid adapter configuration file an Exception will be thrown.

Config File Path
: Determines where the adapter configuration file is located. The configuration file contains the entire information about the log file and how it will be processed.

Debug
: Checking this parameter causes more information to be logged.

Configuring the TDIOutputter

In order to configure the adapter file to use the TDIOutputter you use Eclipse's GLA user interface (the Eclipse GLA plug-in). Below a description of how to configure the outputter using the Eclipse user interface:

1. Open the adapter configuration file for editing. Now the Eclipse plug-in is showing the contents of the configuration file.

2. Go to Adapter -> Configuration -> Context Instance.
3. Right click on Context Instance.
4. Choose Add -> Logging Agent Outputter. Now you are able to see and configure the outputter.
5. For the outputter type choose undeclared.
6. Type a description of your choosing in the Description field.
7. Right click on the Outputter and choose Add -> Property.
8. Name the property "*tdi_correlation_id*".
9. For the value of this property use an arbitrary and **unique** value which will become the correlation ID of the GLA Connector using this configuration file.
10. If no value is filled the TDIOutputter will use a default value and will register any Connector which attempts to start its adapter configuration file.
11. Go to Adapter -> Contexts -> Basic Context Implementation and right click over it.
12. Choose add -> Logging Agent Outputter.
13. Fill the name and description fields.
14. In the Executable Class field enter "com.ibm.di.connector.gla.TDIOutputter".
15. Make sure the role is set to outputter.
16. In the Role version field add a number (for example: 1.0.0).
17. For the unique ID click browse and choose the outputter you just made in steps 2 – 7.
18. Now you have configured the outputter and you are ready to use this adapter configuration file with the GLA Connector.

Using the Connector

To configure the GLA Connector you must have a valid adapter configuration file. The path to the file must be set in the Connector **Config File Path** parameter.

When the GLA Connector starts, a GLA instance is started in a separate thread inside the Connector. Starting the adapter in a separate thread makes it possible to start iterating through the entries before GLA has completed processing the entire log file. When the Connector receives CBE objects it stores them into a queue, which orders the elements in FIFO (first-in-first-out) manner. When there are no elements in the queue and the Connector wants to take an element from it, it will not return null value but will wait until an element is available (that is, it blocks). On the other side of the queue when it is full and an element needs to be added to it, it will block until there is available space in the queue.

Conditions like end-of-data, GLA adapter errors etc. are handled by special messages in the queue, enabling the Connector to work in the manner expected of a Tivoli Directory Integrator Connector.

When iterating, CBE objects are read one by one from the queue, and delivered to the GLA Connector. The CBE object itself is stored into an Attribute called "rawCBEObject" of the work Entry. The CBE attributes are also set in the work Entry.

In order to be able to handle the situation when more than one Connector instance is running simultaneously a mechanism to send the correct CBE objects to the correct GLA Connector is required. This is achieved by using a unique correlation ID parameter in the TDIOutputter configuration. Before a GLA Connector starts the adapter configuration file it gets the correlation ID from the TDIOutputter configuration (the Connector actually parses the adapter configuration file). Then it registers it in an internal TDIOutputter table. When the TDIOutputter is ready to send the generated CBE object it gets its correlation ID and takes the GLA Connector which is registered with this ID in the table.

Schema

The unprocessed, raw CBE object read from the TDIoutputter queue object is available in the following attribute, ready to be mapped into the *work* entry:

Table 11.

Attribute Name	Description
$rawCBE	This attribute holds a single CBE object which is a result of the processed application log file. The number of the CBE objects depends on the configuration of the parser in the adapter configuration file.

The remaining attributes follow the specification as outlined in the output map schema definition in the documentation for the "CBE Parser" on page 297.

See also

The example demonstrating the processing of a DB2 log file, in the *TDI_install_dir*/examples/glaconnector directory,

"RAC Connector" on page 241,

"CBE Function Component" on page 395,

GLA Users Guide.

HTTP Client Connector

The HTTP Client Connector enables greater control on HTTP sessions than the URL Connector provides. With the HTTP Connector you can set HTTP headers and body using predefined attributes. Also, any request to a server that returns data is available for the user as attributes.

This Connector supports secure connections using the SSL protocol when so requested by the server, for example when accessing a server using the '`https://`' prefix in an URL. If client-side certificates are required by the server, you will need to add these to the Tivoli Directory Integrator truststore, and configure the truststore in `global.properties` or `solution.properties`. More information about this can be found in the *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*, in the section named "Client SSL configuration of TDI components".

Note: The HTTP Client Connector does not support the Advanced Link Criteria (see "Advanced link criteria" in *IBM Tivoli Directory Integrator V7.1 Users Guide*).

Modes

The HTTP client Connector can be used in four different AssemblyLine modes. These are:

Iterator
: Each call to the Connector requests the same URL configured for the Connector. This causes the Connector to run forever requesting the same page unless you include a Parser in the Connector's configuration. If you include a Parser, the Parser notifies when the last entry has been read from the connection and the Connector eventually causes an AssemblyLine to stop.

Lookup
: In this mode the Connector requests a page every time the Lookup function is called. In your search criteria you can specify the page or URL to request, or include any number of parameters all of which are appended to the base URL as request parameters.

AddOnly
: In this mode the Connector request is performed much like the Iterator mode.

Call/Reply
: In this mode the Connector has two attribute maps, Input and Output. When the AssemblyLine invokes the Connector, an Output map operation is performed, followed by an Input map operation.

Lookup Mode

In Lookup mode you can dynamically change the request URL by setting the search criteria as follows:

- If you have only one criteria and the attribute is named **url**, then the value specified in the criteria is used as the request URL.

  ```
  url equals $url
  ```

- If you have more than one criteria or the only criteria is anything but **url**, then all attribute names and values are appended to the URL given by the Connector configuration as the request URL.

  Base URL: http://www.example_page_only.com/lookup.cgi

  Search Criteria:

  ```
  name equals john
  mail equals doe.com
  ```

  Resulting URL: http://www.example_page_only.com/lookup.cgi?name=john&mail=doe.com

- The Lookup function ignores the operand. So if you specify **contains** instead of **equals** the Connector still constructs the URL as if equals where used.

Special attributes

When using the Connector in Iterator or Lookup mode the following set of attributes or properties is returned in the Connector ("*conn*") entry:

http.responseCode
The HTTP response code as an Integer object.

200 OK —-> 200

http.responseMsg
The HTTP response message as a String object.

200 OK —-> OK

http.content-type
The content type for the returned http.body (if any).

http.content-encoding
The encoding of the returned http.body (if any).

http.content-length
The number of bytes in http.body.

http.body
This object is an instance/subclass of java.io.InputStream class that can be used to read bytes of the returned body.

```
var body = conn.getObject ("http.body");
var ch;

while ( (ch = body.read()) != -1 ) {
 task.logmsg ("Next character: " + ch);
}
```

Consult the Javadocs for the InputStream classes and their methods.

http.body.response
When the Connector operates in AddOnly mode, responses from the server http.body part will be made available in this Attribute; and the http.body Attribute as it was on the outbound call will be unmodified. If on the outbound call you did not specify a value in the http.body Attribute, then on return from the server the http.body Attribute will be identical to the http.body.response Attribute.

http.text-body
If the http.content-type starts with the sequence text/, the Connector assumes the body is textual data and reads the http.body stream object into this attribute.

When using the Connector in AddOnly mode the Connector transmits any attribute named **http.** as a header. Thus, to set the content type for a request name the attribute **http.content-type** and provide the value as usual. One special attribute is **http.body** that can contain a string or any java.io.InputStream or java.io.Reader subclass.

For all modes the Connector always sets the **http.responseCode** and **http.responseMsg** attributes. In AddOnly mode this is special because the **conn** object being passed to the Connector is the object being populated with these attributes. To access these you must obtain the value in the Connector's **After Add** hook.

Character Encoding

The HTTP Client Connector uses internally the HTTP Parser to parse the input and output streams of the created socket to the specified URL. The default character encoding used for this is ISO-8859-1.

If the HTTP Client Connector has a configured Parser, then this parser is used to write the `http.body` attribute using its specified character encoding; or, if not specified, the default character encoding of the platform is used.

To explicitly specify the character encoding of the `http.body` attribute, use the **Content Type** parameter of the HTTP Client Connector. For more information see "Configuration."

Configuration

The Connector has the following parameters:

HTTP URL
The HTTP page to request.

Note: If you use an https:// address, you might need to import a certificate as well.

Request Method
The HTTP method to use when requesting the page. See http://www.w3.org/Protocols/HTTP/Methods.html for more information

Username
If set the HTTP Authorization header is set using this parameter along with the **Password** parameter.

Password
Used if **Username** is specified.

Proxy If specified, connect to a proxy server rather than directly to the host specified in the URL. The format is *proxyhost*:*port* (for example, `proxy:8080`), where `proxy` is the name of the *proxyhost*, and `8080` is the *port* number to use.

File to HTTP Body
The full path of the file. The file contents are copied as HTTP body in the HTTP message. This overrides any possible Parser processing.

Content Type
If set, this will be used as the *http.content-type* for the file sent as specified by the **File to HTTP Body parameter**, or other *HTTP Body Attribute* that may be present in the Entry (see the HTTP Attributes described above).

File from Response HTTP Body
The full path of the file. The body of the response HTTP message is copied to the file.

Timeout
Timeout in seconds for each of the operations: connecting to the server and receiving response from it. A timeout of zero is interpreted as an infinite timeout. If the timeout expires, a `java.net.SocketTimeoutException` is raised (for more information see the online documentation for `java.net.Socket`).

Detailed Log
If this parameter is checked, more detailed log messages are generated.

You select a Parser from the Parser pane; select the parser by clicking the top-left **Select Parser** button. If specified, this Parser is used to generate the **http.body** content when sending data. The parser gets an entry with those attributes where the name does not begin with **http.** Also, this Parser (if specified) gets the **http.body** for additional parsing when receiving data. However, do not specify system:/Parsers/ibmdi.HTTP, because a message body does not contain another message.

Examples

In your attribute map you can use the following assignment to post the contents of a file to the HTTP server:

```
// Attribute assignment for "http.body"
ret.value = new java.io.FileInputStream ("myfile.txt");

// Attribute assignment for "http.content-type"
ret.value = "text/plain";
```

The Connector computes the **http.content-length** attribute for you. There is no need to specify this attribute.

See also

"URL Connector" on page 285,
"HTTP Server Connector" on page 115,
"HTTP Parser" on page 319.

Old HTTP Client Connector

Note: This Connector is kept for legacy purposes only. If you are setting up a new Connector, please use the HTTP Client Connector instead. It is also deprecated, and will be removed in a future version of Tivoli Directory Integrator.

The Old HTTP Client Connector enables greater control on HTTP sessions than the URL Connector provides. With the HTTP Connector you can set HTTP headers and body using predefined attributes. Also, any request to a server that returns data is available for the user as attributes.

Note: The Old HTTP Client Connector does not support the Advanced Link Criteria (see "Advanced link criteria" in *IBM Tivoli Directory Integrator V7.1 Users Guide*).

Modes

The Old HTTP Client Connector can be used in three different AssemblyLine modes. These are:

Iterator
Each call to the Connector requests the same URL configured for the Connector. This causes the Connector to run forever requesting the same page unless you include a Parser in the Connector's configuration. If you include a Parser, the Parser notifies when the last entry has been read from the connection and the Connector eventually causes an AssemblyLine to stop.

Lookup
In this mode the Connector requests a page every time the Lookup function is called. In your search criteria you can specify the page or URL to request, and include any number of parameters all of which are appended to the base URL as request parameters.

AddOnly
In this mode the Connector request is performed much like the Iterator mode.

Lookup Mode

In Lookup mode you can dynamically change the request URL by setting the search criteria as follows:

- If you have one and only one criteria and the attribute is named **url** then the value specified in the criteria is used as the request URL.

  ```
  url equals $url
  ```

- If you have more than one or the only criteria is anything but url, then all attribute names and values are appended to the URL given by the Connector configuration as the request URL.

  Base URL: http://www.example_name.com/lookup.cgi

  Search Criteria:

  ```
  name equals john
  mail equals doe.com
  ```

  Resulting URL: http://www.example_name.com/lookup.cgi?name=john&mail=doe.comThe Lookup function ignores the operand. So if you specify *contains* instead of *equals* the Connector still constructs the URL as if equals where used.

Special attributes

When using the Connector in Iterator or Lookup mode the following set of attributes or properties is returned in the Connector entry:

http.responseCode
The HTTP response code as an Integer object.

200 OK —-> 200

http.responseMsg
The HTTP response message as a String object.

200 OK —-> OK

http.content-type
The content type for the returned http.body (if any).

http.content-encoding
The encoding of the returned http.body (if any).

http.content-length
The number of bytes in http.body.

http.body
This object is an instance/subclass of java.io.InputStream class that can be used to read bytes of the returned body.

```
var body = conn.getObject ("http.body");
var ch;

while ( (ch = body.read()) != -1 ) {
 task.logmsg ("Next character: " + ch);
}
```

Consult the Javadocs for the InputStream classes and their methods.

http.text-body
If the http.content-type starts with the sequence **text/**, the Connector assumes the body is textual data and reads the http.body stream object into this attribute.

When using the Connector in AddOnly mode the Connector transmits any attribute named **http.** as a header. Thus, to set the content type for a request, name the attribute **http.content-type** and provide the value as usual. One special attribute is **http.body** that can contain a string or any java.io.InputStream or java.io.Reader subclass.

For all modes the Connector always sets the **http.responseCode** and **http.responseMsg** attributes. In AddOnly mode this is a bit special since the **conn** object being passed to the Connector is the object being populated with these attributes. To access these you must obtain the value in the Connector's **After Add** hook.

Configuration

The Connector has the following parameters:

HTTP URL
The HTTP page to request.

Request Method
The HTTP method to use when requesting the page. See http://www.w3.org/Protocols/HTTP/Methods.html for more information.

Username
If set, the HTTP Authorization header uses this parameter along with the **Password** parameter.

Password
Used if **Username** is specified.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Parser If specified, this Parser is used to generate the posted data for an **Add** operation.

Examples

In your attribute map you can use the following assignment to post the contents of a file to the HTTP server:

```
// Attribute assignment for "http.body"
ret.value = new java.io.FileInputStream ("myfile.txt");

// Attribute assignment for "http.content-type"
ret.value = "text/plain";
```

The Connector computes the **http.content-length** attribute for you. There is no need to specify this attribute.

See also

"URL Connector" on page 285,
"Old HTTP Server Connector" on page 121,
"HTTP Client Connector" on page 107.

HTTP Server Connector

IBM Tivoli Directory Integrator provides a HTTP Server Connector that listens for incoming HTTP connections and acts like a HTTP server. Once it receives the request, it parses the request and sends the parsed request to the AssemblyLine workflow to process it. The result is sent back to the HTTP client. By default, the returned result has a content-type of "text/html".

The Connector supports Server and Iterator Modes. Server mode is the recommended mode:

- In Server Mode, when the connection is accepted the Connector clones the AssemblyLine, and hands the request off to the clone. The parent process resumes waiting for new incoming connections.
- In Iterator Mode, this cloning does not happen and the request is handled in the current AssemblyLine itself. However, once the request has been processed, the connection (and hence, the AssemblyLine) will end. This may not suit your purpose.

If a Parser is specified then the Connector processes **post** requests and parses the contents using the specified Parser; **get** requests do not use the Parser. If a **post** request is received and no Parser is specified the contents of the **post** data is returned as an attribute (**postdata**) in the returned entry.

The HTTP Server Connector uses ibmdi.HTTP as internal Parser if no Parser is specified.

The HTTP Server Connector supports Server mode only.

The Connector parses URL requests and populates an entry in the following manner:

http://localhost:8888/path?p1=v1&p2=v2

```
http.method : 'GET'
http.Host   : 'localhost:8888'
http.base   : '/path'
http.qs.p1  : 'v1'
http.qs.p2  : 'v2'
```

http://localhost:8888/?p1=v1&p2=v2

```
http.method : 'GET'
http.Host   : 'localhost:8888'
http.base   : '/'
http.qs.p1  : 'v1'
http.qs.p2  : 'v2'
```

If a **post** request is used then it is expected that the requestor is sending data on the connection as well. Depending on the value for the **Parser** parameter the Connector performs the following actions:

Parser present
: Instantiates the Parser with the HTTP input stream. Connector delegates getNext to the Parser's getEntry and returns whatever the Parser returns.

Parser not present
: Puts contents of post data in a Connector attribute called **http.body**.

The session with the HTTP client is closed when the Connector receives a getNext request from the AssemblyLine and there is no more data to fetch. For example, if the Parser has returned a null value, or on the second call to getNext if no Parser is present.

Connector structure and workflow

The HTTP Server Connector receives HTTP requests from HTTP clients and sends HTTP responses back. As mentioned above, the default content-type header is set to "text/html"; you can override that by setting the Entry attribute `http.content-type` to the appropriate value before the Connector returns the result to the client.

After the AssemblyLine initializes the HTTP Server Connector, it calls the `getNextClient()` method of the Connector. This method blocks until a client request arrives. When a request is received (and Server Mode is selected), the Connector creates a new instance of itself, which is handed over to the AssemblyLine that subsequently spawns a new AssemblyLine thread for that Connector instance. This design feature provides the ability to process each Event in a separate thread, which allows the HTTP Server Connector to process several HTTP events in parallel. The AssemblyLine then calls the `getNextEntry()` method on this new Connector instance in the new thread. Each Entry returned by the `getNextEntry()` call represents an individual HTTP request from the HTTP client. The Connector's `replyEntry(Entry conn)` method is called for each Entry returned from `getNextEntry()` to send to the client the corresponding HTTP response.

Connector Client Authentication

The parameter HTTP Basic Authentication governs whether client authentication will be mandated for HTTP clients accessing this connector over the network.

There are two different ways to implement HTTP Basic Authentication with the HTTP Server Connector:

1. **Using an Authentication Connector**

   This is a mechanism for backward compatibility with the old HTTP EventHandler (which is no longer present in Tivoli Directory Integrator 7.1). A connector parameter **Auth Connector** specifies a Tivoli Directory Integrator Connector that will be used in Lookup Mode, with the username and password for the HTTP Basic Authentication data specified as the Link Criteria:

   - If the lookup returns an Entry, the authentication is considered successful and the HTTP Server Connector proceeds with processing the client's request.
   - If the lookup cannot find an Entry, the client is not authenticated and the request will not be processed.

2. **Script authentication**

   This mechanism requires a certain amount of coding, but provides more power and lets you implement authentication through your own scripting. It can only be used when the **Auth Connector** parameter is NULL or empty.

   The Connector will make available to you the username and password values in the "After Accepting connection" Hook through the `getUserName()` and `getPassword()` public Connector methods. It is now your responsibility to implement the authentication mechanism. You should call the Connector's `rejectClientAuthentication()` method from the AssemblyLine hook if authentication is not successful. Consider the following example authentication script code:

```
var httpServerConn = conn.getAttribute("connectorInterface").getValue(0);
var username = httpServerConn.getUserName();
var password = httpServerConn.getPassword();

//perform verification here
successful = true;

if (!successful) {
 httpServerConn.rejectClientAuthentication();
}
```

Chunked Transfer Encoding

When the parameter **Chunked Transfer Encoding** is enabled, the Connector will write the HTTP body as series of chunks.

When chunked encoding is used, you are responsible for calling the Connector's `putEntry(entry)` method for each chunk – the value of the "http.body" Attribute of the Entry provided will be sent as an HTTP chunk. The `replyEntry(entry)` Connector's method is automatically called by the AssemblyLine at the end of the iteration – it will write the last chunk of data (if the "http.body" Attribute is present) and close the chunk sequence.

When a Parser is specified to the HTTP Server Connector, it will be the stream returned by the Parser that will be sent as a HTTP chunk on each `putEntry(entry)` or `replyEntry(entry)` call.

Configuration

The Connector needs the following parameters:

TCP Port
The TCP port to listen for incoming requests (the default port is **80**).

Connection Backlog
This represents the maximum queue length for incoming connection indications (a request to connect). If a connection indication arrives when the queue is full, the connection is refused.

Content Type
The default HTTP content type to use for outbound data. This value is overridden by the "http.content-type" Attribute of the `work` object. The default is text/html.

TCP Data as Properties
If the check box is checked (default), the TCP connection properties are accessed through the getProperty() method of the `conn` Entry object. If unchecked, the TCP connection properties appear as Entry Attributes.

HTTP Headers as Properties
If the check box is checked, all HTTP headers are accessible using the getProperty() method of the `conn` Entry object. If unchecked, all HTTP headers appear as Entry Attributes.

HTTP Basic Authentication
If enabled (by default it is not), clients will be challenged for HTTP Basic authentication.

Auth Realm
The authentication realm sent to the client when requesting HTTP Basic authentication. The default is "IBM Tivoli Directory Integrator".

Auth Connector
This drop-down list specifies an Authenticator Connector. If a Connector is specified it must exist in the Connector library and be configured for Lookup mode.

When this parameter is specified, the HTTP Server Connector will issue authentication requests to any client (for example, a Web browser) that tries to access this service and does not provide authentication data. When the client provides the username/password the HTTP Server Connector will call the Authenticator Connector's lookup method providing the username and password attributes. Hence, the authentication Connector must be configured using a Link Criteria where the $username and $password attributes are used. A typical link criteria would be:

```
username equals $username
password equals $password
```

If the search fails, the HTTP Server Connector denies the request and sends an authentication request back to the client. If the search succeeds, the HTTP Server Connector processes the request.

The entry returned by the authenticator Connector can be accessed through the "auth.entry" Property of the event Entry.

For more details on client authentication, and for an alternative method to using an Auth Connector, see “Connector Client Authentication” on page 116.

Use SSL
If enabled (by default it is not), then the Connector will require clients to use SSL; non-SSL connection requests will fail.

When SSL is used, the Connector will use the default Tivoli Directory Integrator Server SSL settings – certificates, keystore and truststore.

Require Client Authentication
If enabled (by default it is not), the Connector mandates client authentication when using SSL. This means that the Connector will require clients to supply client-side SSL certificates that can be matched to the configured Tivoli Directory Integrator trust store. This parameter is only taken into account if the previous parameter (Use SSL) is enabled as well.

Chunked Transfer Encoding
If checked, the HTTP body of the message is transferred as a series of chunks; see "Chunked Transfer Encoding" on page 116.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Note: You can select a Parser from the **Parser** configuration pane; click on the *Inherit from:* button in the bottom right corner when the Parser pane is active.

Connector Schema

Listed below are all the Attributes supported by the HTTP Server Connector.

Input Attributes

- http.* - Any HTTP header.
- http.Authorization - The type of http authorization.
- http.base - HTTP base parameter.
- http.body - The body of HTTP request.
- http.content-length - The number of bytes in http.body.
- http.content-type - The type of HTTP content, for example text/plain, text/xml, etc.
- http.method - HTTP method type. Valid values are: GET/POST/PUT
- http.qs.* - Query string parameter.
- http.remote-pass - The remote user password.
- http.remote-user - The remote username.
- auth.entry - The entry returned by the authenticator Connector.
- tcp.inputstream - Socket input stream.
- tcp.outputstream - Socket output stream.
- tcp.remoteIP - Remote IP address.
- tcp.remotePort - Remote port.
- tcp.remoteHost - Remote host name.
- tcp.localIP - Local IP address.
- tcp.localPort - Local port.
- tcp.localHost - Local host name.
- tcp.socket - Raw socket object.

Output Attributes

- http.body - The body of HTTP response.
- http.content-type - The type of HTTP content.
- http.redirect - Redirect client to specified location.
- http.status - Status code of returned operation.

See also

"URL Connector" on page 285,
"HTTP Client Connector" on page 107,
"HTTP Parser" on page 319.

Old HTTP Server Connector

Note: This Connector is kept for legacy purposes only. If you are creating a new Connector, please use HTTP Server Connector instead. It is also deprecated, and will be removed in a future version of Tivoli Directory Integrator.

The Old HTTP Server Connector listens for incoming HTTP connections and returns the **get** parameters as an entry. If a Parser is specified then the Connectors process **post** requests and parse the contents using the specified Parser. **get** requests do not use the Parser. If a **post** request is received and no Parser is specified, the contents of the **post** data are returned as an attribute (**postdata**) in the returned entry.

The Connector parses URL requests and populates an entry in the following manner:

http://host/path?p1=v1&p2=v2

```
entry.path = "/path"
entry.p1="v1"
entry.p2="v2"
```

http://host?p1=v1&p2=v2

```
entry.path="/"
entry.p1="v1"
entry.p2="v2"
```

If a **POST** request is used then it is expected that the requestor is sending data on the connection as well. Depending on the value for the **Parser** parameter the Connector performs the following actions:

Parser present
Instantiates the Parser with the HTTP input stream. Connector delegates getNext to the Parser's getEntry and returns whatever the Parser returns.

Parser not present
Puts contents of post data in an attribute called **postdata**.

```
entry.postdata = "postdata"
```

The session with the HTTP client is closed when the Connector receives a getNext request from the AssemblyLine and there is no more data to retrieve. For example, if the Parser has returned a null value, or on the second call to getNext no Parser is present. If you call getNext (for example, iterate) after having received a null from the Connector.

Configuration

The Connector needs the following parameters:

TCP Port
The TCP port to listen to (the default port is **80**).

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Parser The name of a Parser to handle the contents of **post** requests.

See also

"URL Connector" on page 285,
"HTTP Server Connector" on page 115.

IBM Tivoli Directory Server Changelog Connector

The IBM Tivoli Directory Server (TDS) Changelog Connector is a specialized instance of the LDAP Connector. The IBM Tivoli Directory Server Changelog Connector contains logic to iterate the Changelog. It returns various attributes, including the **changes** attribute. This attribute is of type `java.lang.String` and contains an incremental LDIF that you can read using the Parser FC and the LDIF Parser. This technique will also retrieve any changes made to the "objectClass" of the changed entry as well (since the objectClass Attribute returned by the Connector is that of the changelog entry itself).

The Connector can be used in batch-oriented runs where it starts at a specific change number and stops after the last **Changelog** entry. It can also be run in continuous mode where you specify the timer values for periodically checking for the next **Changelog** entry.

The Connector reads Changelog entries and automatically increases the Changelog counter by one for each iteration. When the Connector tries to read a non-existing Changelog entry, the Connector goes to sleep for a period of time (**Sleep Interval**). If the total time the Connector is waiting for a new entry exceeds the **Timeout** value, then the Connector returns to the caller with a **null** value (end of iteration).

This connector also exposes a "**Use Notifications**" option which specifies whether the Connector will use a polling or a notification mechanism to retrieve new IDS changes. If set to *false* the Connector will poll for new changes. If this parameter is set to *true* then after processing all existing changes the Connector will block and wait for an unsolicited event notification from the IBM Directory Server. The Connector will not sleep and timeout when the notification mechanism is used.

This connector also supports Delta Tagging, at the Entry level, the Attribute level and the Attribute Value level. It is the LDIF Parser that provides Delta support at the Attribute and Attribute Value levels.

The Connector will detect *modrdn* operations in the Server's changelog, see “Detect and handle modrdn operation” on page 182 for more information.

Note: This component is not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.

Attribute merge behavior

In older versions of Tivoli Directory Integrator, in the IBM Tivoli Directory Server Changelog Connector merging occurs between Attributes of the changelog Entry and changed Attributes of the actual Directory Entry. This creates issues because you cannot detect the attributes that have changed. The Tivoli Directory Integrator 7.1 version of the Connector has logic to address these situations, configured by a parameter: **Merge Mode**. The modes are:

- **Merge changelog and changed data** - The Connector merges the attributes of the Changelog Entry with changed attributes of the actual Directory Entry. This is the older implementation and keeps backward compatibility.
- **Return only changed data** - Returns only the modified/added attributes and makes Changelog Iterator and Delta mode easier. This is the default; note that in configurations developed under and migrated from earlier versions of Tivoli Directory Integrator, you may need to select **Merge changelog and changed data** manually so as to ensure identical behavior.
- **Return both** - Returns an Entry that contains changed attributes of the actual Directory Entry and an additional attribute called "changelog" that contains attributes of the Changelog Entry. Allows you to easily distinguish between two sets of Attributes.

Delta tagging is supported in all merge modes and entries can be transferred between different LDAP servers without much scripting.

Differences between changelog on distributed TDS and z/OS TDS

There are some differences in the way the changes to password policy operational attributes are logged to `cn=changelog` in IBM Tivoli Directory Server on z/OS and in Distributed IBM Tivoli Directory Server (which runs on other platforms). The currently known differences in behavior are listed below:

1. Modify of userpassword

   A modify operation to change the `userpassword` will remove attributes such as `pwdfailuretime`, `pwdreset`, `pwdaccountlockedtime`, `pwdgraceusetime` and `pwdexpirationswarned` from an entry in the directory. It will also update the `pwdchangedtime`.

   Distributed TDS records these updates in the LDIF along with the replace of the `userpassword` value in the changelog entry.

   z/OS TDS only records the replace of the `userpassword` in the LDIF, omitting the generated deletion of the operational attributes.

   A password change can also conditionally update the `pwdhistory` attribute of an entry. We know that this change is not logged in z/OS TDS. Although we have no test data to show that it is indeed logged in Distributed TDS, we suspect it is.

2. Password value in the changelog LDIF

   z/OS TDS suppresses the actual value (for security reasons) and instead displays the value as "`userpassword: *ComeAndGetIt*`".

   Distributed TDS shows the `userpassword` value as is. Note that we only have test output where password encryption is not being used, and thus the actual password is displayed "in the clear". If password encryption is active, probably the tagged, encrypted value is shown.

3. Add of a user entry

   An add operation of a user entry containing a password will conditionally add the `pwdreset` attribute with a value of true if the effective policy for the user indicates this to be the case for new entries.

   Distributed TDS includes "`PWDRESET: true`" in the changelog entries LDIF for the add, but z/OS TDS does not.

4. Authentication via a grace login

   When a password is expired, but "grace" logins are allowed, authentication (via either a bind or compare operation) succeeds and an additional value of the attribute `pwdgraceusetime` is added to the user entry. Distributed TDS records this as a single value added to the entry. z/OS TDS records this as a replace of the entire set of values for the `pwdgraceusetime` attribute, listing all the old values and the one new one.

Configuration

The Connector needs the following parameters:

LDAP URL
: The LDAP URL for the connection (ldap://*host*:*port*).

Login username
: The LDAP distinguished name used for authentication to the server. Leave blank for anonymous access.

Login password
: The credentials (password).

Authentication Method
: Type of LDAP authentication. Can be one of the following:

  - **Anonymous** - If this authentication method is set then the server, to which a client is connected, does not know or care who the client is. The server allows such clients to access data configured for non-authenticated users. The Connector automatically specifies this authentication method if no username is supplied. However, if this type of authentication is

chosen and **Login username** and **Login password** are supplied, then the Connector automatically sets the authentication method to Simple.

- **Simple** - using **Login username** and **Login password**. Treated as anonymous if **Login username** and **Login password** are not provided. Note that the Connector sends the fully qualified distinguished name and the client password in cleartext, unless you configure the Connector to communicate with the LDAP Server using the SSL protocol.
- **CRAM-MD5** - This is one of the SASL authentication mechanisms. On connection, the LDAP Server sends some data to the LDAP client (that is, this Connector). Then the client sends an encrypted response, with password, using MD5 encryption. After that, the LDAP Server checks the password of the client. CRAM-MD5 is supported only by LDAP v3 servers. It is not supported against any supported versions of Tivoli Directory Server.
- **SASL** - The client (this Connector) will use a Simple Authentication and Security Layer (SASL) authentication method when connecting to the LDAP Server. Operational parameters for this type of authentication will need to be specified using the **Extra Provider Parameters** option; for example, in order to setup a DIGEST-MD5 authentication you will need to add the following parameter in the Extra Provider Parameters field:

  ```
  java.naming.security.authentication:DIGEST-MD5
  ```

  For more information on SASL authentication and parameters see: http://java.sun.com/products/jndi/tutorial/ldap/security/sasl.html.

  Note: Not all directory servers support all SASL mechanisms and in some cases do not have them enabled by default. Check the documentation and configuration options for the directory server you are connecting to for this information.

Use SSL
If Use SSL is **true** (that is, checked), the Connector uses SSL to connect to the LDAP server. Note that the port number might need to be changed accordingly.

ChangeLog Base
The search base where the Changelog is kept. The standard DN for this is **cn=changelog**.

Extra Provider Parameters
Allows you to pass a number of extra parameters to the JNDI layer. It is specified as name:value pairs, one pair per line.

Iterator State Key
Specifies the name of the parameter that stores the current changelog number in the User Property Store of the IBM Tivoli Directory Integrator, to allow processing to stop and begin again at the last processed change. This must be a unique name for all parameters stored in one instance of the IBM Tivoli Directory Integrator User Property Store. If this value is left blank, the connector will start processing at the beginning of the changes with each AssemblyLine restart. The **Delete** button deletes this information from the User Property Store.

Start at changenumber
Specifies the starting changenumber (default value:1) Each Changelog entry is named **changenumber=intvalue** and the Connector starts at the number specified by this parameter and automatically increases by one. The special value **EOD** means start at the end of the Changelog. This parameter is only used when the Iterator State is blank or not saved.

The **Query** button retrieves the first and last change numbers from the Server.

State Key Persistence
Governs the method used for saving the Connector's state to the System Store. The default (and recommended setting) is **End of Cycle**, and choices are:

After read
Updates the System Store when you read an entry from the directory server's change log, before you continue with the rest of the AssemblyLine.

End of cycle
Updates the System Store with the change log number when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual
Switches off the automatic updating of the System Store with this Connector's state information; instead, you will need to save the state by manually calling the IBM Directory Server Changelog Connector's *saveStateKey()* method, somewhere in your AssemblyLine.

Merge Mode
Governs the method used for merging attributes of the Changelog Entry and changed attributes of the actual Directory Entry. The default is **Return only changed data**, and choices are:

Merge changelog and changed data
The Connector merges the attributes of the Changelog Entry with changed attributes of the actual Directory Entry. This option selects the behavior of older versions of Tivoli Directory Integrator and maintains backwards compatibility.

Return only changed data
Returns only the modified or added attributes.

Return both
Returns changed attributes of the actual Directory Entry, plus an additional attribute called "changelog" that contains an Entry with changelog attributes.

Use Notifications
Specifies whether to use notification when waiting for new changes in IBM Directory Server. If enabled, the Connector will not sleep or timeout (and the corresponding parameters are ignored), but instead wait for a Notification event from the IBM Directory Server.

Batch retrieval
Specifies how searches are performed in IDS changelog. When unchecked, the Connector will perform incremental lookups (backward compatible mode). When checked, and the server supports "Sort Control", searches will be preformed with query 'changenumber>=*some_value*', corresponding to the last retrieval you made; this works in conjunction with the next parameter, **Page Size**. By default, this option is unchecked.

Page Size
Specifies the size of the pages IDS will return entries on (default value is 500). It is used only when **Batch retrieval** is set to *true*, that is, checked.

Timeout
Specifies the number of seconds the Connector waits for the next Changelog entry. The default is 0, which means wait forever.

Sleep Interval
Specifies the number of seconds the Connector sleeps between each poll. The default is 60.

Detailed Log
If this field is checked, additional log messages are generated.

Note: Changing Timeout or Sleep Interval values will automatically adjust its peer to a valid value after being changed (for example, when timeout is greater than sleep interval the value that was not edited is adjusted to be in line with the other). Adjustment is done when the field editor looses focus.

See also

Enabling change log on IBM Tivoli Directory Server,
"LDAP Connector" on page 181,

"Active Directory Change Detection Connector" on page 8,
"Sun Directory Change Detection Connector" on page 215,
"z/OS LDAP Changelog Connector" on page 291.

ITIM Agent Connector

The ITIM Agent Connector uses the IBM Tivoli Identity Manager's JNDI driver to connect to ITIM Agents (the JNDI driver uses the DAML protocol). Thus the ITIM Agent Connector is able to connect to all ITIM Agents that support the DAML protocol.

The Connector itself does not understand the particular schema of the ITIM Agent it is connected to – it provides the basic functionality to create, read, update and delete JNDI entries.

The ITIM Agent Connector supports the Iterator, Lookup, AddOnly, Update and Delete modes.

This Connector uses the client library `enroleagent.jar` from the ITIM 4.6 release.

Setting up SSL for the ITIM Agent Connector

Since the `enroleagent.jar` client library uses JSSE (Java based keystore/truststore) for SSL authentication, you are now required to mention the SSL-related certificate details in the global.properties/ solution.properties; previous versions of the ITIM Agent Connector required you to specify the certificate name in the "CA Certificate File" parameter. You need to first import the ITIM Agent's certificate into the Tivoli Directory Integrator truststore.

For example, with the following command you import the `servercertificate.der` file into `tim.jks`.

```
keytool –import –file servercertificate.der –keystore tim.jks
```

After you import the certificate, you need to mention this truststore in the "server authentication" section of the global.properties /solution.properties file.

```
## server authentication

javax.net.ssl.trustStore=E:\IBMDirectoryIntegrator\tim.jks
{protect}-javax.net.ssl.trustStorePassword=<jks_keystore_password>
javax.net.ssl.trustStoreType=jks
```

Note: The "CA Certificate File" property of the ITIM Agent Connector is no longer present, since now the certificates mentioned in the JKS trust store in global.properties or solution.properties are being used.

Configuration

The Connector needs the following parameters:

Agent URL
: The URL used to connect to the ITIM Agent, in the form "https://<agent_ip_address>:<port>", for example "https://localhost:45580"

UserName
: The username specified in the configuration of the ITIM Agent – used by the Connector to authenticate to the ITIM Agent.

Password
: The password specified in the configuration of the ITIM Agent – used by the Connector to authenticate to the ITIM Agent.

Connection Retry Count
: Specifies how many times to retry a failed connection (including initial connection attempt). If no value is specified the ITIM JNDI driver uses a default value of 3.

Search Filter
: Filter expression to use in Iterator mode. If no value is specified a default filter of "(objectclass=*)" is used to return all Entries.

Detailed Log
Checking this parameter generates extra log messages.

Known Issues

The Connector has been briefly tested with a few ITIM Agents. Some lookup issues have been detected that result from constraints of the underlying Agents implementation:

Sometimes simple JNDI searches might not return the expected results. For example, if you are using the Windows 2000 Agent, the JNDI search for the Guest user account "(eruid=Guest)" might return more than one Entry; or when you are using the Red Hat Linux Agent the search for the "root" group "(erLinuxGroupName=root)" returns an empty result set.

A work-around for these cases is to use an extended search filter where the object class is specified: "(&(eruid=)(objectclass=<classname>))". So for the Windows 2000 Agent the search would look like "(&(eruid=Guest)(objectclass=erW2KAccount))" and for the Red Hat Linux Agent the search filter should be "(&(eruid=root)(objectclass=erLinuxGroup))".

This work-around does not work for all lookup issues, for example the search for the Windows "Administrators" group (Windows 2000 Agent) – "(erW2KGroupName=Administrators)" returns an empty result set. The extended search filter "(&(eruid=Administrators)(objectclass=erW2KGroup))" returns an empty result set too.

When you encounter a lookup problem:

1. Make sure you are using the latest version of the Agent.
2. Try the work-around described above.
3. If the work-around doesn't work, examine the schema of the Agent for other attributes that can be used for Entry identification.

Here are a few examples for how other attributes from the Agent schema can be used for Entry identification:

- In the search for the Windows "Administrators" group mentioned above, instead of "erW2KGroupName" attribute, the attribute "erW2KGroupCommonName" could be used. The filter "(erW2KGroupCommonName=Administrators)" works fine and you will get the "Administrators" group Entry.
- For the LDAP-X Agent, searches for LDAP users ("erXLdapAccount" class) with the default "eruid" attribute might fail – in this case you can use the "cn" attribute for Entry identification.

See also

DAML/DSML Protocol,
“ITIM DSMLv2 Connector” on page 83.

IBM MQ Connector

The IBM MQ Connector is a specialized instance of the "JMS Connector" on page 151.

JDBC Connector

The JDBC Connector provides database access to a variety of systems. To reach a system using JDBC you need a JDBC driver from the system provider. This provider is typically delivered with the product in a jar or zip file. These files must be in your path or copied to the `jars/` directory of your Tivoli Directory Integrator installation; otherwise you may get cryptic messages like "Unable to load T2 native library", indicating that the driver was not found on the classpath.

You will also need to find out which of the classes in this jar or zip file implements the JDBC driver; this information goes into the **JDBC Driver** parameter.

The JDBC Connector also provides multi-line input fields for the SELECT, INSERT, UPDATE and DELETE statements. When configured, the JDBC connector will use the value for any of these instead of its own auto-generated statement. The value is a template expanded by the parameter substitution module that yields a complete SQL statement. The template has access to the connector configuration as well as the *searchcriteria* and *conn* objects. The *work* object is not available for substitution, since the connector does not know what *work* contains. Additional provider parameters are also supported in the connector configuration.

The JDBC Connector supports the following modes: AddOnly, Update, Delete, Lookup, Iterator, Delta.

This Connector in principle can handle secure connections using the SSL protocol; but it may require driver-specific configuration steps in order to set up the SSL support. Refer to the manufacturer's driver documentation for details.

Connector structure and workflow

The JDBC connector makes a connection to the specified data sources during the connector initialization. While making a connection to the specified data source extra provider parameters are checked for, and set if they are specified. The auto-commit flag setting is also handled and set during connection initialization.

The JDBC connector builds SQL statements internally using a predefined mapping table. The connector flow behaves the same way as other connectors in AddOnly, Update, Delete, Iterator and Lookup modes.

In addition, this Connector supports Delta mode; the delta functionality for the JDBC connector is handled by the ALComponent (a generic building block common to all Connectors). The ALComponent will do a lookup and apply the delta Entry to a target Entry before doing an update, and then decide what the correct database operation must be. The Connector will then use the SQL statements for add, modify or delete, corresponding to what the operation is.

Understanding JDBC Drivers

In order for the JDBC Connector to access a relational database, it needs to access a *driver*, a set of subroutines or methods contained in a Java classlibrary. This library must be present in the *CLASSPATH* of Tivoli Directory Integrator, otherwise Tivoli Directory Integrator will not be able to load the library when initializing the Connector, and hence be unable to talk to the Relational Database (RDBMS). A good way to install a JDBC driver library such that Tivoli Directory Integrator can use it is to copy it into the *TDI_install_dir*`/jars` directory, or a directory of your choosing subordinate to this, for example *TDI_install_dir*`/jars/local`.

Notes:

1. Some drivers may contain native code, typically presented in .dll or .so files – these need to be added to the PATH variable in order for Tivoli Directory Integrator to pick them up at run time.
2. Be aware of duplicate class names. If your libraries contain classes that duplicate classes in any of the other libraries in the CLASSPATH, it is undefined which class will be loaded.
3. The library should be readable by all users.

4. The applications wishing to use the library must be restarted after installing the library (Configuration Editor, Tivoli Directory Integrator Servers.)

There are 4 fundamental ways of accessing an RDBMS through JDBC (these are often referred to as driver types):

1. Drivers that implement the JDBC API as a mapping to another data access API, such as Open Database Connectivity (ODBC). Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC Bridge driver is an example of a Type 1 driver; this driver is generally part of the JVM, so it does not need to be specified separately on the Tivoli Directory Integrator classpath.

   To configure ODBC, see "Specifying ODBC database paths" on page 138.

   Note: The JDBC-ODBC bridge may be present in any of the different platform-dependent JVM's that IBM ships with the product. However, IBM supports the JDBC-ODBC bridge on Windows platforms only. In addition, performance is likely to be sub-optimal compared to a dedicated, native ("Type 4") driver. Commercial ODBC/JDBC bridges are available. If you need an JDBC-ODBC bridge, consider purchasing a commercially available bridge; see also the JDBC-ODBC bridge drivers discussion at http://java.sun.com/products/jdbc/driverdesc.html..
2. Drivers that are written partly in the Java programming language and partly in native code. The drivers use a native client library specific to the data source to which they connect. Again, because of the native code, their portability is limited.
3. Drivers that use a pure Java client and communicate with a middleware server using a database-independent protocol. The middleware server then communicates the client's requests to the data source.
4. Drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

With the exception of the JDBC-ODBC bridge on Windows, we only use Type 4 drivers with IBM Tivoli Directory Integrator. We will discuss other types as well—in the context of each of the supported databases—for a better understanding.

JDBC Type 3 and Type 4 drivers use a network protocol to communicate to their back-ends. This usually implies a TCP/IP connection; this will either be a straight TCP/IP socket, but if the driver supports it, it can be a Secure Socket Layer (SSL) connection.

Note: When working with custom prepared statements, make sure that the JDBC used driver is compliant with JDBC 3.0. There is a known issue with IBM SolidDB 6.5, since the driver implements only JDBC 2.0. If the **Use custom SQL prepared statements** option is enabled when working with this database, a `java.lang.NullPointerException` will be thrown.

Connecting to DB2

The IBM driver for JDBC and SQLJ bundled with Tivoli Directory Integrator was obtained from http://www-306.ibm.com/software/data/db2/java. It is JDBC 1.2, JDBC 2.0, JDBC 2.1 and JDBC 3.0 compliant.

Information about the JDBC driver for IBM DB2® is available online; a starting point and example for configuration purposes is the section on "How JDBC applications connect to a data source" in the DB2 Developer documentation. This driver may or may not suit your purpose.

Driver Licensing

This driver does not need further licensing for DB2 database systems (that is, the appropriate license file, `db2jcc_license_cu.jar` is already included), except DB2 for z/Series and iSeries. In order for the driver to be able to communicate with the latter two systems you would need to obtain the DB2 Connect product, and copy its license file, `db2jcc_license_cisuz.jar`, to the *jars/3rdparty/IBM* directory. In addition, since this driver is a FAT client with natively compiled code (.dll/.so), the DB2 Connect install path needs to be added to the PATH variable for these libraries to be used.

Based on the JDBC driver architecture DB2 JDBC drivers are divided into four types.

1. DB2 JDBC Type 1

   This is an DB2 ODBC (not JDBC) driver, which you connect to using a JDBC-ODBC bridge driver. This driver is essentially not used anymore.

   A JDBC Type 1 driver can be used by JDBC 1.2 JDBC 2.0, and JDBC 2.1.

   To configure ODBC, see "Specifying ODBC database paths" on page 138.

2. DB2 JDBC Type 2

   The DB2JDBC Type 2 driver is quite popular and is often referred to as the *app* driver. The app driver name comes from the notion that this driver will perform a native connect through a local DB2 client to a remote database, and from its package name (COM.ibm.db2.jdbc.app.*).

   In other words, you have to have a DB2 client installed on the machine where the application that is making the JDBC calls runs. The JDBC Type 2 driver is a combination of Java and native code, and will therefore usually yield better performance than a Java-only Type 3 or Type 4 implementation.

   This driver's implementation uses a Java layer that is bound to the native platform C libraries. Programmers using the J2EE programming model will gravitate to the Type 2 driver as it provides top performance and complete function. It is also certified for use on J2EE servers.

   The implementation class name for this type of driver is *com.ibm.db2.jdbc.app.DB2Driver*.

   The JDBC Type 2 drivers can be used to support JDBC 1.2, JDBC 2.0, and JDBC 2.1.

3. DB2 JDBC Type 3

   The JDBC Type 3 driver is a pure Java implementation that must talk to middleware that provides a DB2 JDBC Applet Server. This driver was designed to enable Java applets to access DB2 data sources. An application using this driver can talk to another machine where a DB2 client has been installed.

   The JDBC Type 3 driver is often referred to as the net driver, appropriately named after its package name (COM.ibm.db2.jdbc.net.*).

   The implementation class name for this type of driver is *com.ibm.db2.jdbc.net.DB2Driver*.

   The JDBC Type 3 driver can be used with JDBC 1.2, JDBC 2.0, and JDBC 2.1.

4. DB2 JDBC Type 4

   The JDBC Type 4 driver is also a pure Java implementation. An application using a JDBC Type 4 driver does not need to interface with a DB2 client for connectivity because this driver comes with Distributed Relational Database Architecture™ Application Requester (DRDA® AR) functionality built into the driver.

   The implementation class name for this type of driver is *com.ibm.db2.jcc.DB2Driver*.

   The latest version of this driver (9.1) supports SSL connections; this requires setting a property in the **Extra Provider Parameters** field. For more information see http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.apdv.java.doc/doc/rjvdsprp.htm. Note that the target database must be set up such that it accepts incoming SSL connections.

If you are running DB2 on a z/OS platform, and the database is not configured correctly with the required stored procedure for retrieving the schema, you might encounter some problems using the JDBC Connector. If the JDBC Connector's query schema throws an exception, or the Add/Update action on JDBC tables fails for BLOB data types, contact your database administrator and request that the required

stored procedure for retrieving the schema be installed. For more information about accessing DB2 from Java, see also Overview of Java Development in DB2 UDB for Linux, UNIX, and Windows.

Connecting to Informix Dynamic Server

If you install the Informix Client SDK, you will also install Informix ODBC drivers which allow you to use a JDBC-ODBC bridge driver. This driver is not recommended for production use. To configure ODBC, see "Specifying ODBC database paths" on page 138.

However, we recommend you use the Informix JDBC driver, version 3.0. It is a pure-Java (Type 4) driver, which provides enhanced support for distributed transactions and is optimized to work with IBM WebSphere® Application Server.

It consists of a set of interfaces and classes written in the Java programming language. Included in the driver is Embedded SQL/J which supports embedded SQL in Java.

The implementation class for this driver is *com.informix.jdbc.IfxDriver*. For information how to install the Informix driver, see http://publib.boulder.ibm.com/infocenter/idshelp/v111/index.jsp?topic=/com.ibm.conn.doc/jdbc_install.htm

Connecting to Oracle

Based on the JDBC driver architecture the following types of drivers are available from Oracle.

1. Oracle JDBC Type 1

   This is an Oracle ODBC (not JDBC) driver, that you connect to using a JDBC-ODBC bridge driver. Oracle does supply an ODBC driver, but does not supply a bridge driver. Instead, you can use the default JDBC-ODBC bridge that is part of the JVM, or get one of the JDBC-ODBC bridge drivers from http://java.sun.com/products/jdbc/drivers.html. This configuration works fine, but a JDBC Type 2 or Type 4 driver will offer more features and will be faster.

   To configure ODBC, see "Specifying ODBC database paths" on page 138.

2. Oracle JDBC Type 2

   There are two flavors of the Type 2 driver.

   - JDBC OCI client-side driver

     This driver uses Java native methods to call entrypoints in an underlying C library. That C library, called OCI (Oracle Call Interface), interacts with an Oracle database. The JDBC OCI driver requires an Oracle client installation of the same version as the driver. The use of native methods makes the JDBC OCI driver platform specific. Oracle supports Solaris, Windows, and many other platforms. This means that the Oracle JDBC OCI driver is not appropriate for Java applets, because it depends on a C library. Starting from Version 10.1.0, the JDBC OCI driver is available for installation with the OCI Instant Client feature, which does not require a complete Oracle client-installation. Please refer to the Oracle Call Interface for more information.

   - JDBC Server-Side Internal driver

     This driver uses Java native methods to call entrypoints in an underlying C library. That C library is part of the Oracle server process and communicates directly with the internal SQL engine inside Oracle. The driver accesses the SQL engine by using internal function calls and thus avoiding any network traffic. This allows your Java code to run on the server to access the underlying database in the fastest possible manner. It can only be used to access the same database.

3. Oracle JDBC Type 4

   Again, there are two flavors of the Type 4 driver.

   - JDBC Thin client-side driver

     This driver uses Java to connect directly to Oracle. It implements Oracle's SQL*Net Net8 and TTC adapters using its own TCP/IP based Java socket implementation. The JDBC Thin client-side driver does not require Oracle client software to be installed, but does require the server to be configured with a TCP/IP listener. Because it is written entirely in Java, this driver is platform-independent.

The JDBC Thin client-side driver can be downloaded into any browser as part of a Java application. (Note that if running in a client browser, that browser must allow the applet to open a Java socket connection back to the server.)

This is the most commonly-used driver. In general, unless you need OCI-specific features, such as support for non-TCP/IP networks, use the JDBC Thin driver.

The implementation class for this driver currently is *oracle.jdbc.driver.OracleDriver*.

- JDBC Thin server-side driver

  This driver uses Java to connect directly to Oracle. This driver is used internally within the Oracle database, and it offers the same functionality as the JDBC Thin client-side driver, but runs inside an Oracle database and is used to access remote databases. Because it is written entirely in Java, this driver is platform-independent. There is no difference in your code between using the Thin driver from a client application or from inside a server.

For more information about accessing Oracle from Java, see also Java, JDBC & Database Web Services, and the Oracle JDBC FAQ.

Connecting to SQL Server

The Microsoft SQL Server 2005 driver for JDBC supports the JDBC 1.22, JDBC 2.0 and JDBC 3.0 specification. It is a Type 4 driver.

The implementation class for this driver is *com.microsoft.sqlserver.jdbc.SQLServerConnection*. It is contained in the driver file `sqljdbc.jar`, typically obtained from the MS SQL Server 2005 installation, at `<Microsoft SQL Server 2005-Install-Dir>\sqljdbc_1.1.1501.101_enu\sqljdbc_1.1\enu\sqljdbc.jar`.

You can also use other third party drivers for connecting to Microsoft SQL Server.

The jTDS JDBC 3.0 driver distributed under the GNU LGPL is a good choice. This is a Type 4 driver and supports Microsoft SQL Server 6.5, 7, 2000, and 2005. jTDS is 100% JDBC 3.0 compatible, supporting forward-only and scrollable/updateable ResultSets, concurrent (completely independent) Statements and implementing all the `DatabaseMetaData` and `ResultSetMetaData` methods. It can be downloaded freely from http://jtds.sourceforge.net. More information about this driver is available from the Web site.

Connecting to Sybase Adaptive Server

The jConnect for JDBC driver by Sybase provides high performance native access (Type 4) to the complete family of Sybase products including Adaptive Server Enterprise, Adaptive Server Anywhere, Adaptive Server IQ, and Replication Server.

jConnect for JDBC is an implementation of the Java JDBC standard; it supports JDBC 1.22 and JDBC2.0, plus limited compliance with JDBC 3.0. It provides Java developers with native database access in multi-tier and heterogeneous environments. You can download jConnect for JDBC quickly, without previous client installation, for use with thin-client Java applications - like IBM Tivoli Directory Integrator.

The implementation class name for this driver is *com.sybase.jdbc3.jdbc.SybDriver*.

You can also use other third party drivers for connecting to Sybase.

The jTDS JDBC 3.0 driver distributed under the GNU LGPL is a good choice. This is a Type 4 driver and supports Sybase 10, 11, 12 and 15. jTDS is 100% JDBC 3.0 compatible, supporting forward-only and scrollable/updateable ResultSets, concurrent (completely independent) Statements and implementing all the `DatabaseMetaData` and `ResultSetMetaData` methods. It can be downloaded freely from http://jtds.sourceforge.net. More information about this driver is available from the Web site.

Connecting to Derby

Derby is a relational database, modeled after IBM DB2, written entirely in Java. This database product as well as its drivers are bundled with Tivoli Directory Integrator. The network driver is a Type 4 driver: native Java code.

The implementation class name for this driver is *org.apache.derby.jdbc.ClientDriver*.

Refer to the Derby Developer's Guide, *Conventions for specifying the database paths*, for more information about how to construct your JDBC URLs when using Derby.

Connecting to IBM SolidDB

IBM SolidDB is a relational in-memory database that offers enhanced performance compared to Derby. Thus, it can be used as System Store instead of the Derby database, to boost the performance of the components relying on it.

The driver provided by IBM SolidDB is Type 4 (completely implemented in Java). It can be obtained from the database installation, from *SolidDB_install_dir*/jdbc/SolidDriver2.0.jar.

Detailed information on SolidDB can be found at http://publib.boulder.ibm.com/infocenter/soliddb/v6r3/index.jsp.

Note: The driver for IBM SolidDB is not JDBC 3.0 compliant, but implements JDBC 2.0 only. This may cause problems if you use Custom Prepared Statements.

Specifying ODBC database paths

When you use ODBC connectivity using the JDBC-ODBC bridge (supported on Windows systems only) you can specify a database or file path the ODBC driver must use, if the ODBC driver permits. This type of configuration avoids having to define a data source name for each database or file path your Connector uses.

jdbcDriver
sun.jdbc.odbc.JdbcOdbcDriver

jdbcSource
jdbc:odbc:*driver name*;DBQ=*path*

The syntax of this parameter is dependent on the following conditions:

MS Access is installed
Open the ODBC data source control panel, and select the **User DSN** tab. In this table you see the driver names you can use in the JDBC Source parameter. For example, if you want to access an MS Access database (`C:\Documents and Settings\`*username*`\My Documents\mydb.mdb`), provide the following value for the JDBC source:

```
jdbc:odbc:MS Access Database;dbq=C:\Documents and Settings\username\My Documents\mydb.mdb
```

MS Access is not installed
If MS Access is not installed, and you are on a Windows system, use the following value:

```
jdbc:odbc:Driver={MS Access Driver
 (*.mdb)};dbq=C:\Documents and Settings\username\My Documents\mydb.mdb
```

Alternatively, use the Windows System DSN utility, available under **Administrative Tools -> Data Sources (ODBC)**. Once you define a System DSN, use a jdbcSource parameter like the this:

```
jdbc:odbc:myDSNNameHere
```

Check the Driver list that you get in the utility. Your JDBC URL must exactly match the wording found in this list.

Schema

In Iterator and Lookup modes the JDBC Connector schema depends on the metadata information read from the database for the table name specified. If no table name is given the schema is retrieved using the SQL Select/Lookup statements (if defined; see "Customizing select, insert, update and delete statements" on page 142).

In AddOnly, Delete, Update and Delta the JDBC Connector schema depends on the metadata information read from the database for the table name specified.

Configuration

The Connector needs the following parameters:

JDBC URL

See documentation for your JDBC provider. Typical URL's for common RDBMS systems are:

Table 12. JDBC URL examples

RDBMS	Example connection URL
IBM DB2 (using the DRDA driver)	"jdbc:db2://hostname:port/dbname"
Informix® Dynamic Server 10.0	"jdbc:informix-sqli://hostname:port/ dbname:informixserver=<Informix Server Name>"
Oracle (using the "thin driver")	"jdbc:oracle:thin:@hostname:1521:SID", using "host:port:sid" syntax, TNSListener accepting connections on port 1521
Microsoft SQL Server (using Microsoft's driver)	"jdbc:sqlserver://hostname:1433;databasename=dbname;", SQL Server listening for connections on port 1433
Sybase 15 (also older versions from v. 10), using jConnect 6.05	"jdbc:sybase:Tds:hostname:port/"
Derby	"jdbc:derby://hostname:port/<server path to database>;options"
IBM SolidDB 6.3	"jdbc:solid://hostname:port"

JDBC Driver

The JDBC driver implementation class name. The default value of `sun.jdbc.odbc.JdbcOdbcDriver` addresses the JDBC-ODBC bridge, which is not recommended for production use. For databases for which another type of driver is available, typical driver implementation class names are:

Table 13. Driver implementation class names

RDBMS	Driver implementation class name
IBM DB2, type 2 or 4	com.ibm.db2.jcc.DB2Driver
Oracle, type 4	oracle.jdbc.driver.OracleDriver
Informix Dynamic Server 10.0	com.informix.jdbc.IfxDriver
Microsoft SQL Server, type 4	com.microsoft.sqlserver.jdbc.SQLServerDriver
Sybase 15 (also older versions from v. 10)	com.sybase.jdbc3.jdbc.SybDriver
Derby	org.apache.derby.jdbc.ClientDriver
IBM SolidDB 6.3	solid.jdbc.SolidDriver

Also see "Understanding JDBC Drivers" on page 133, and "Database Connector" on page 39.

Username

Signon to the database using this username; only the tables accessible to this user will be shown.

Password

The password used in the signon for the user.

Schema
The schema from the table of the database that you want to use. If left blank, the value of the jdbcLogin (that is, the **Username** parameter) is used.

Note: Throughout the Tivoli Directory Integrator documentation, you will find the term Schema used to mean the data definition of the object you are accessing. However, in the RDBMS world, the term Schema has a different meaning, namely the overall collection of data definitions, tables and objects grouped under one identifier (username). For this particular parameter in this particular Connector, we use it in the RDBMS sense.

Table Name
The table or view to operate on. This is only used when the Connector operates in Lookup or Update mode. If the **SQL Select** parameter is not specified, then the Iterator mode Connector also uses this parameter to construct a default SELECT statement.

Select...
Click this button to bring up a list of available table names that you can select from to enter into the **Table Name** field. This only works if the underlying database supports this; for example, Microsoft Access using ODBC does not.

Return null values
If the checkbox is enabled then null valued attributes return an empty value. If left unchecked then the defined null behavior is followed. The default null behavior will remove attributes that receive null.

Commit
Controls when database transactions are committed. Options are:

- **After every database operation** (default)
- **After every database operation (Including Select)**
- **On Connector close**
- **Manual**
- **End of Cycle**

Manual means user must call the *commit()* method of the JDBC Connector—or *rollback()*, as appropriate.

Note: The option **After every database operation (Including Select)** has been provided for those databases which lock database tables in transactions even when they only have been Selected for read operations (notably DB2).

Use Prepared Statements
The value of this check box determines whether to use PreparedStatement or Statement. If this is selected PreparedStatement will be used by the JDBC connector, else Statement will be used. The default is checked, that is, *true*, meaning try to pre-compile SQL statements, fall back to normal.

Use custom SQL prepared statements
Specifies whether custom specified SQL statements are prepared statements (true) or not (false). The default is unchecked, that is, false. Also see “Option to turn off Prepared Statements in the JDBC connector” on page 144.

SQL Select
The select statement to execute when selecting entries for iteration, that is, Iterator mode. If you leave this blank, the default construct (SELECT * FROM TABLE) is used. See “Customizing select, insert, update and delete statements” on page 142.

The button marked "**...**" to the right of the **SQL Select** parameter presents a Link Criteria dialog where you can fill out the link criteria form and generate the proper SQL Where clause.

Use the **Add** button to add more rows to build your selection criteria. The **Match any** checkbox will generate an OR expression rather than the default AND expression. Note that this is a one

way helper: anything you already have in the SQL select parameter will be replaced by the generated expression. If the SQL select parameter contains a "where" clause, then only the Where-clause is replaced.

SQL Lookup
The custom SQL statement to use for lookups (used in Lookup, Update and Delete modes).

SQL Insert
The custom SQL statement to use when inserting into the database, using AddOnly or Delta mode.

SQL Update
The custom SQL statement to use when updating the database, using Update or Delta mode.

SQL Delete
The custom SQL statement to use when using Delete or Delta mode.

Alter Session Statements
This parameter is a multi-line field where you can specify ALTER SESSION commands. The following is an example of an ALTER SESSION command:

```
"SET NLS_FORMAT 'YYYY-MM-DD'"
```

Extra Provider Parameters
Additional JDBC provider parameters (`name:value` - one for each line). With this you can specify additional parameters supported by the JDBC provider. You should check your driver documentation for the supported parameters and then use them. For example, specific to DB2:

```
securityMechanism:KERBEROS_SECURITY
loginTimeout:20
readOnly:true
```

Date Format
A format string used to parse dates when they are supplied as strings. You can select from a list of pre-defined format strings, or supply your own.

Disable padding for Insert
The value of this check box determines whether the padding should be disabled for Insert operations in certain modes. By default, this checkbox is off, which means padding is **not** disabled. In other words, it is enabled for Insert operations in AddOnly, Update and Delta modes.

Disable padding for Update
The value of this check box determines whether the padding should be disabled for Update operations in certain modes. By default, this checkbox is off, which means padding is **not** disabled. In other words, it is enabled for Update operations in Update and Delta modes.

Disable padding for Lookup
The value of this check box determines whether the padding should be disabled for Lookup operations in certain modes. By default, this checkbox is off, which means padding is **not** disabled. In other words, it is enabled for Lookup operations in Lookup, Update, Delete and Delta modes.

Auto-create table
When the connector is configured in one of the output modes (AddOnly, Update) you have this additional option in the advanced section.

The **Auto-create table** option will make the connector create a simple table based on the attribute map and schema for the connector. This is only done when the table does not exist in the database.

When auto-creating a table the connector will first derive the column names from the attribute map. If the attribute map is empty, the schema is used to get the list of column names. Once the column names are determined a SQL *CREATE TABLE* statement is generated with each of the column names. If the schema has a definition for the column name it will be consulted to

determine the syntax for the column. There are two parts in the schema that will determine the syntax for the column. First, if the "Native Syntax" is specified it is used as-is. Next, if there is no native schema provided the connector uses the "Java Class" to derive the syntax. The Java-class field in the schema should specify any of the following values:

Table 14. Java class to SQL type mapping

Value	Generated SQL type
Integer or java.lang.Integer	INT
String or java.lang.String	VARCHAR(255)
Double or java.lang.Double	DOUBLE
Date or java.util.Date	TIMESTAMP

If there is no schema information about the column, or if the value is not recognized the connector will use "VARCHAR(255)" in the generated create table statement.

Connector Flags
A list of flags to enable specific behavior.

{ignoreFieldErrors}

If getting field values causes an error, this flag causes the Connector to return the Java exception object as the value instead of throwing the exception (that is, calling the Connectors *Fail EventHandlers).

Detailed Log
If this field is checked, additional log messages are generated.

Link Criteria configuration
Link criteria specified in the Connector's configuration for Lookup, Delete, Update and Delta modes are used to specify the WHERE clause in the SQL queries used to interact with the database.

The Tivoli Directory Integrator operand **Equal** is translated to the equal sign (=) in the SQL query, while the **Contains**, **Start With** and **End With** operators are mapped to the **like** operator.

Skip Lookup in Update or Delete mode
The JDBC Connector supports the **Skip Lookup** general option in Update or Delete mode. When it is selected, no search is performed prior to actual update and delete operations. Special code in the Connector retrieves the proper number of entries affected when doing update or delete.

Customizing select, insert, update and delete statements

Overview
The JDBC connector has the ability to expand a SQL template before executing any of its SQL operations. There are five operations where the templates can be used. These operations are:

Table 15. SQL Operations

Operation	Connector Parameter name	Description	Mode(s)
SELECT	SQL Select	Used in Iterator mode (no search criteria).	Iterator
INSERT	SQL Insert	Used when adding an entry to the data source.	Update, AddOnly, Delta
UPDATE	SQL Update	Used when modifying an existing entry in the data source.	Update, Delta
DELETE	SQL Delete	Used when deleting an existing entry in the data source.	Delete, Delta

Table 15. SQL Operations (continued)

Operation	Connector Parameter name	Description	Mode(s)
LOOKUP	SQL Lookup	A SELECT statement with a WHERE clause. Used when searching the data source.	Lookup, Delete, Update

If the template for a given operation is not defined (for example, null or empty), the JDBC connector will use its own internal template.

When there is a template defined for an operation, the template must generate a complete and valid SQL statement. The template can reference the standard parameter substitution objects (for example, `mc`, `config`, `work`, `Connector`), as well as the JDBC schema for the table configured for the connector and a few other convenience objects.

Note: The template for the LOOKUP operation can contain a WHERE clause filtering the elements that will be returned by the query. But when the connector is in Lookup, Update or Delete mode the **Link Criteria** parameter is mandatory, as it is used to assemble a WHERE clause for the executed query. If **Link Criteria** is omitted an exception will be thrown:

```
java.lang.Exception: CTGDIS143E No criteria can be built from input (no link criteria specified).
    at com.ibm.di.server.SearchCriteria.buildCriteria(Unknown Source)
```

Therefore if a configuration is created and it uses a WHERE clause in the LOOKUP template the you must provide a **Link criteria** although one will not be needed. The connector will simply ignore it and the template query will be used. In order to save you from adding unneeded "dummy" conditions in the **Link criteria**,the solution is to check the option **Build criteria from custom script** and leave the displayed script area empty.

Metadata Object

The information about JDBC field types is provided as an Entry object named `metadata`. Each attribute in the metadata Entry object corresponds to a field name and the value will be that field's corresponding type. For example, a table with the following definition:

```
CREATE TABLE SAMPLE (
  name varchar(255),
  age numeric(10),
)
```

could be referenced in the following manner, during parameter substitution:

```
{javascript<<EOF
      metadata = params.get("metadata");
      if (metadata.getAttribute("name").equals("varchar"))
   return "some sql statement";
     else
  return "some other sql statement";
EOF
}
```

Link Object (Link Criteria)

The LinkCriteria values are available in the *link* object. The link object is an array of link criteria items. Each item has fields that define the link criteria according to configuration. If the configured link criteria is defined as *cn equals john doe* then the template could access this information with the following substitution expressions:

```
link[0].name ► "cn"
link[0].match ► "="
link[0].value ► "john doe"
link[0].negate ► false
```

A complete template for a SELECT operation could look like this:

```
SELECT * FROM {config.jdbcTable} WHERE {link[0].name} = '{link[0].value}'
```

Convenience Objects

Generating the WHERE clause or the list of column names is not easy without resorting to JavaScript code. As a convenience, the JDBC Connector makes available the column names that would have been used in an UPDATE and INSERT statement as columns; this does not apply to SELECT and LOOKUP statements. This value is a comma-delimited list of column names. The textual WHERE clause is available as "whereClause" to simplify operations. Below is an example of how to use both:

```
SELECT {columns} from {config.jdbcTable} WHERE {whereClause}
```

for example, *SELECT a,b,c from TABLE-A WHERE a > 1 AND b = 2*

Table 16. Information available for different statements

Object	SELECT	LOOKUP	INSERT	DELETE	UPDATE
config	yes	yes	yes	yes	yes
Connector	yes	yes	yes	yes	yes
metadata	no	maybe	maybe	yes	yes
conn	no	no	yes	yes	yes
columns	no	no	yes	yes	yes
link	no	yes	no	yes	yes
whereClause	no	yes	no	yes	yes

Option to turn off Prepared Statements in the JDBC connector

The JDBC connector uses *PreparedStatement* to efficiently execute an SQL statement on a connected RDBMS server. However, there maybe cases when the JDBC driver may not support PreparedStatements. As a fall back mechanism a config parameter (jdbcPreparedStatement, labelled **Use prepared statements** in the configuration panel) is available in the configuration of the JDBC connector. The config parameter is a Boolean flag that indicates whether the JDBC connector should use PreparedStatements. If this is set the connector will use PreparedStatement and will fall back to normal *Statements* (java.sql.Statement) in case of an exception. If this is not set, normal Statement will be used by the JDBC connector while executing SQL queries. This checkbox gives an option to a Tivoli Directory Integrator solution developer to handle situations when there are problems due to use of PreparedStatements. The checkbox will be set by default, meaning that the JDBC connector will use PreparedStatement.

The `findEntry`, `putEntry`, `deleteEntry` and the `modEntry` methods of the JDBC connector check for the value of usePreparedStatement flag to determine whether to use PreparedStatements or Statements. If a connector config does not have this flag (as in an older version of the config), the value of this param will be *true* by default. This ensures that there are no migration issues or impact.

Custom Prepared Statements

When you use the JDBC connector without custom SQL statements, it uses *Prepared Statements* internally for faster access to the JDBC target database. The connector builds simple SQL prepared statements to perform the connector operations (for example, `SELECT * from TABLE WHERE x = ?`) and then uses the JDBC API to provide values for the placeholders (the question marks) in the statement. This makes it easy to provide a variety of Java objects to the database without having to do complex string encoding of values.

Every now and then, the user needs to override the SQL statements the JDBC connector creates. This is where the SQL Insert/Update/etc. configuration parameters come into play. The user can specify the exact SQL statement used for a specific operation. The SQL statement is provided to the JDBC driver as a standalone statement, which basically means that the SQL statement contains values for columns used in the statement. This also means that the user must build the statement including values for columns in the configuration itself, including any complex string encoding of values.

With the custom prepared statements feature, the user can now use proper prepared statements. This will make custom statements much easier since it removes the encoding requirement. Also, if the statement does not change between calls, the prepared statement is reused, which results in faster execution.

The JDBC connector has a parameter named **Use custom prepared statements**. The checkbox is to enable/disable prepared statements and is false by default. When this checkbox is enabled you must use proper syntax in the custom SQL field. While you can still use constants in the SQL statement you must either properly escape any question marks in the statement or provide an expression for the prepared statement placeholder. Type "?" or use ctrl-<space> to bring up the code completion helper that lets you choose from the list of attributes you have in the output map as well as other common expressions. Once you have chosen an expression and press **Enter** the editor will insert that string between two question marks. Also note the syntax highlights that will provide feedback as to what is being interpreted as a placeholder expression:

```
Select * from table where modified_date > ?{javascript return new java.util.Date()}
 and something_else < ?{conn.a}
```

Note that it is also possible to use expressions in the statement where you normally don't have placeholders. Prepared Statements can also be provided by means of an API; sometimes this may be the easiest option to use. See “APIs to allow specification of Prepared Statements” on page 147 for more information. If you use the JDBC Connector with a JDBC 2.0 driver, in particular with IBM solidDB, also see “Connecting to IBM SolidDB” on page 138.

Background:

To more fully explain the need for the above functionality, consider the following:

The current format for custom SQL statements requires the user to enter a complete SQL statement. Often, this can be tedious as well as near impossible if the values are complex or binary in nature. An option is present in the JDBC connector (**Use custom prepared statements**) where the user can toggle between Prepared Statement mode and the current plain string mode. Prepared statement mode applies a different syntax to the custom SQL statements. When prepared statement mode is chosen, no substitution is done to the string as is the case when non prepared statement mode is selected.

As an example let's use a simple SELECT statement to illustrate the problem with building a complete SQL statement:

```
SELECT * FROM TABLE_NAME WHERE modified_date > 03/04/09
```

This statement contains a where clause filtering on modifed_date being greater than a given date. This example shows the problematic nature of SQL statements: Is the date march 3rd 2009 or april 4th 2009? There are other cases where building a complete SQL statement becomes even more problematic.

With the **Use custom prepared statements** option, the user can use a slightly modified SQL prepared statement. A SQL prepared statement in JDBC terms is a complete SQL statement with placeholders for values. The placeholder is the question mark and is replaced with a value at runtime.

```
SELECT * FROM TABLE_NAME WHERE modified_date > ?
```

However, the JDBC connector needs to know which value to provide for each placeholder. To make the prepared statement as syntactically correct as possible while also providing the ability to specify which values are provided at runtime, the prepared statement syntax is slightly modified:

```
SELECT * FROM TABLE_NAME WHERE modified_date > {expression}
```

This is not a valid prepared statement syntax, but the JDBC connector will parse this string and replace "{expression}" with a single question mark before executing the statement. The "{expression}" is a Tivoli

Directory Integrator expression that provides the value for the prepared statement placeholder. The text field editors for the custom SQL statements provide additional functionality to aid the user in building the statement.

Note: When custom prepared statements are used, the user must also provide the WHERE clause where applicable.

Additional JDBC Connector functions

Apart from the standard functions exposed by all Connectors, this Connector also exposes several other functions you can use in your scripts. You could call them using the special variable *thisConnector*, for example, `thisConnector.commit();` — when called from any scripting location in the Connector.

commit()
Commits any pending database operations.

execSQL (string)
Starts an arbitrary SQL command. Returns the error string if it fails.

execSQLSelect (string)
Starts SQL SELECT command. Returns the error string if it fails.

getNextSQLSelectEntry ()
Having started execSQLSelect you can use this method to get the next entry from the result set.

The Connector's **Table Name** parameter must be empty for this to work correctly.

rollback()
Backs out any database operations performed since the last *commit()* (irrespective of whether the commit was done manually, or as a result of autocommit operations).

The above functions do not interfere with the normal flow of entries and attribute mappings for the Connector.

API to disable or enable parameter substitution

The above subsections describe how the JDBC Connector has access to its parameter substitution capabilities in insert, update, and delete SQL commands that are executed by the Connector. In certain cases, this causes issues because your customized SQL can end up with substrings (starting with a "{" and ending with a "}") that will be acted upon by the parameter substitution mechanism, and should not. The JDBC Connector exposes an API so you can disable or enable parameter substitution for the SQL statements that will be executed by the JDBC Connector.

```
/**
 * set enableParamSubstitute parameter
 *
 *
 */
public void setParameterSubstitution(boolean val)
{
 enableParamSubstitute = val;
}

/**
 * Returns value of enableParamSubstitute parameter
 *
 *
 */
public boolean getParameterSubstitution()
{
  return enableParamSubstitute ;
}
```

An alternative to using this API to avoid unwanted parameter substitution is using escape characters.

The escape character is a "\". If a "\" is encountered in the character directly preceding a {ArgumentIndex} and {TDIReference} (that is, \{ArgumentIndex}and \{TDIReference}), then the parameter substitution will not take place (will not be processed). Instead, the escape character will be removed and the parameter substitution will not occur. For example, \{TDIReference} would simply be {TDIReference} after being processed.

APIs to allow specification of Prepared Statements

For power-users, it may be easier to just use an API to specify the correct Prepared Statement that they would like to use and also all the values that should be used. The following new methods have been added to the JDBCConnector:

```
public PreparedStatement setPreparedModifyStatement(String preparedSql)
public PreparedStatement setPreparedDeleteStatement(String preparedSql)
public PreparedStatement setPreparedInsertStatement(String preparedSql)
public PreparedStatement setPreparedFindStatement(String preparedSql)
public PreparedStatement setPreparedSelectStatement(String preparedSql)
```

With these methods, the user can have code like this to for example do a special select:

```
  ps = thisConnector.connector.setPreparedSelectStatement("Select * from tableName where fieldName = ? and field2= ?")
 ps.setInteger(1, someValue)
 ps.setObject(2, someObject)
```

The Javadocs for the methods give more examples.

Timestamps

If you want to store a timestamp value containing both a date and a time, you must make sure you provide an object of type **java.sql.Timestamp**, as you can with this Attribute Mapping:

```
ret.value = java.sql.Timestamp(java.util.Date().getTime());
```

The **java.sql.Timestamp** type can also come in handy if for some reason storing DATE fields in tables causes trouble, for example the Oracle error **ORA-01830: date format picture ends before converting entire input string**. Normally, if you try to store date/time values which are in the form of strings, the **Date Format** parameter comes into play to convert the string into the DATE type the underlying database expects, and if there is a mismatch between this parameter and your date/time value formatted as a string, problems will ensue.

To troubleshoot your problem:

- What is your Data Pattern configuration?
- Find out how Tivoli Directory Integrator sees this field (check in the schema tab of the Connector). A fair guess is that your JDBC driver will convert the Oracle Data type into a java.sql.TimeStamp or java.sql.Date type (and note that there are differences between java.util.Date and java.sql.Date, in terms of precision amongst others). For example, in the case of a java.sql.Timestamp type, try specifying the construct mentioned above, that is

  ```
  ret.value = java.sql.Timestamp(java.util.Date().getTime());
  ```

  and see if this helps. If it does, then you will be able to use

  ```
  ret.value = java.sql.Timestamp(system.parseDate(work.getString("yourDate"),
              "yyyyMMddHHmmssz").getTime());
  ```
- If none of the above helps, turn the Connector into detailed log mode and see whether the Connector is able to get the schema from the database. If not, the Connector does not use prepared statements which makes it less efficient and more error-prone - so you'll have to make sure that the Connector's **schema** configuration parameter is set correctly.

Padding

Traditionally, the JDBC Connector would pad data to be added in the CHAR datatype column if the length of data was less than the column width. This was the default behavior and there was no option for configuring the padding.

With the advent of the UTF-8 character set, this could result in unexpected behavior since the Connector was not able to determine the exact length of UTF-8 data. This, in turn, resulted in the adding of an indeterminate amount of whitespaces, and the data length became bigger than the column width which resulted in an exception thrown from the database.

To get around this problem we provide you with the option of optionally disable padding for various operations performed by the Connector, namely Insert, Update and Lookup operations, in AddOnly, Lookup, Update, Delete and Delta modes. See the “Configuration” on page 139 section for the parameters selecting this functionality.

For UTF-8 data the padding should be disabled. For Latin-1 characters the padding can be enabled or disabled.

Calling Stored Procedures

The JDBC Connector's "getConnection()" method gives you access to the JDBC Connection object created when the connector has successfully initialized.

In other words, if your JDBC connector is named DBconn in your AL,

```
var con = DBconn.getConnector().getConnection();
```

will give you access to the JDBC Connection object (an instance of java.sql.Connection).

Note: When called from anywhere inside the connector itself, you can also use the *thisConnector* variable.

Here is a code example illustrating how you can invoke a stored procedure on that database:

```
// Stored procedure call
command = "{call DBName.dbo.spProcedureName(?,?)}";

try {
    cstmt = con.prepareCall(command);

    // Assign IN parameters (use positional placement)
    cstmt.setString(1, "Christian");
    cstmt.setString(2, "Chateauvieux");

    cstmt.execute();

    cstmt.close();
    // Tivoli Directory
Integrator will close the connection, but you might want to force a close now.
    DBConn.close();
}

catch(e) {
    main.logmsg(e);
}
```

SQL Databases: column names with special characters

If you have columns with special characters in their names and use the AddOnly or Update modes:

1. Go to the attribute map of the Update or AddOnly Connector
2. Rename the Connector attribute (not the work attribute!) from **name-with-dash** to **"name-with-dash"** (add quotes).

The necessity of using this functionality might be dependent on the JDBC driver you are using, but standard MS Access 2000 has this problem.

Using prepared statements

This section describes how the Connector creates SQL queries. You can skip this section unless you are curious about the internals.

For a database, the Connector uses prepared statements or dynamic query depending on the situation:

- If the Connector gets the schema definition from the database, it uses prepared statements. Also see "Option to turn off Prepared Statements in the JDBC connector" on page 144.
- Otherwise, the Connector creates a dynamic SQL query.

On Multiple Entries

See Appendix B, "AssemblyLine Flow Diagrams," on page 533 for more information about what happens when a Connector has a link criteria returning multiple entries.

For the JDBC Connector in Delete or Update mode, if you have used the setCurrent() method of the Connector and not added extra logic, all entries matching the link-criteria are deleted or updated.

Additional built-in reconnect rules

The JDBC Connector takes advantage of the Reconnect engine that is part of IBM Tivoli Directory Integrator 7.1. In addition to the standard behavior this engine provides, the JDBC Connector has a number of additional built-in rules. The Connector specific built-in rules will perform a reconnect if a java.sql.SQLException is thrown and the exception contains the following messages, evaluated using Regular Expressions:

- ^I/O.*
- ^Io.*
- ^IO.*
- ^ORA-01089.*
- ^Closed Connection.*

These rules are visible in the **Connection Errors** pane in the Connector's configuration.

See also

"Database Connector" on page 39

JMS Connector

Introduction

"JMS" means Java Message Service, and the JMS Connector is a connector that can tap into message queues implemented using the JMS standard. You can learn more about JMS in Sun Microsystem's JMS Tutorial, and read about the API in the JMS specification and API documentation.

The JMS Connector's functions and features are:

- Enables communication of native Entry objects to be passed using a Java Message Service product.
- Supports JMS message headers and properties.
- Supports sending different types of data on the JMS bus (text message, object message, bytes message).
- Allows users to write their own Java code (JMS initiator class) to connect to different JMS systems.
- Allows users to write JavaScript to connect to different JMS systems.
- Support for plugging in other message queues than IBM MQ.
- Supports auto acknowledge and manual acknowledge through the `acknowledge()` method.

The JMS Connector provides access to JMS based systems such as IBM MQ Server or the bundled MQe. A partly-preconfigured version of this Connector exists under the name "**IBM MQ Connector**", where the JMS Server Type is hidden, and pre-set to "IBMMQ".

Refer to Specific topics to see what you might need to do to your IBM Tivoli Directory Integrator installation to make the JMS Connector work.

The Connector enables communication of both native Entry objects and XML text to be passed using a Java Message Server product.

The JMS Connector supports JMS message properties. Each message received by the JMS Connector populates the conn object with properties from the JMS message (see the getProperty() and setProperty() methods of the entry class to access these). **conn** object properties are prefixed with **jms.** followed by the JMS message property name. The property holds the value from the JMS message. When sending a message the user can set properties which are then passed on to the JMS message sent. The JMS Connector scans the **conn** object for properties that starts with **jms.** and set the corresponding JMS message property from the **conn** property.

- JMS: correlationID=12 ——> **conn** jms.correlationID=12
- **conn**:jms.inReplyTo=12 ——> JMS:inReplyTo=12

The **conn** object is only available in a few hooks. See "Conn object" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.

JMS message flow

Everything sent and received by the JMS Connector is a JMS message. The JMS Connector converts the IBM Tivoli Directory Integrator Entry object into a JMS message and vice versa. Each JMS message contains predefined JMS headers, user defined properties and some kind of body that is either text, a byte array or a serialized Java object.

There exists a method as part of the JMS Connector which can greatly facilitate communication with the JMS bus: acknowledge(). The method acknowledge() is used to explicitly acknowledge all the JMS session's consumed messages when **Auto Acknowlege** is unchecked. By invoking acknowledge() of the Connector, the Connector acknowledges all messages consumed by the session to which the message was delivered. Calls to acknowledge are ignored when **Auto Acknowlege** is checked.

Careful thought must be given to the acknowledgement of received messages. As described, the best approach is to not use **Auto Acknowledge** in the JMS Connector, but rather insert a Script Connector right after the JMS Connector in the AssemblyLine, invoking the acknowledge() method of the JMS Connector. This ensures that the window between the relevant message information in the system store being saved, and the JMS queue notification is as small as possible. If a failure occurs in this window, the message is received once more.

Conversely, relying on **Auto Acknowledge** creates a window that exists from the point at which the message is retrieved from the queue (and acknowledged), until the message contents mapped into the entry is secured in the system store. If a failure occurs in this window, the message is lost, which can be a greater problem.

Note: There could be a problem when configuring the JMS Connector in the Config Editor when **Auto Acknowledge** is on, because as long as this is the case, when going through the process of schema discovery using either **Schema->Connect->GetNext** or Quick Discover from Input Map the message will be grabbed and consumed (that is, gone from the input queue). This may be an unintended side-effect. To avoid this, turn **Auto Acknowledge** off before Schema detection — but remember to switch it back on again afterwards, if this is the desired behavior

WebSphere MQ and JMS/non-JMS consumers of messages

When the JMS Connector sends messages to WebSphere MQ, it is capable of sending these messages in two different modes depending on the client which will read these messages:

- the messages are intended to be read by non-JMS clients (the default)
- the messages are intended to be read by JMS clients

By default the Connector sends the messages so that they are intended to be read by non-JMS clients. The major difference between these two modes is that when the messages are intended to be read by non-JMS clients, the JMS properties are ignored. Thus a subsequent lookup on these properties will not find a match.

In order to switch to the "intended to be read by JMS clients" mode, the "Specific Driver Attributes" parameter value must contain the following line (apart from any other attributes specified):
`mq_nonjms=false`

JMS message types

The JMS environment that enables you to send different types of data on the JMS bus. This Connector recognizes three of those types. The three types are referred to as Text Message, Bytes Message and Object Message. The most open-minded strategy is to use Text Message (for example, `jms.usetextmessages=true`) so that applications other than IBM Tivoli Directory Integrator can read messages generated by the JMS Connector.

When you communicate with other IBM Tivoli Directory Integrator servers over a JMS bus the BytesMessage provides a very simple way to send an entire Entry object to the recipient. This is also particularly useful when the entry object contains special Java objects that are not easy to represent as text. Most Java objects provide a toString() method that returns the string representation of it but the opposite is very rare. Also, the toString() method does not always return very useful information. For example, the following is a string representation of a byte array:

`"[B@<memory-address>"`

Text message

A text message carries a body of text. The format of the text itself is undefined so it can be virtually anything. When you send or receive messages of this type the Connector does one of two things depending on whether you have specified a Parser:

- When you specify a Parser the Connector calls the Parser to interpret the text message and return these attributes along with any headers and properties. When sending a message the provided **conn** object is

passed to the Parser to generate the text body part. This makes it easy to send data in various formats onto a JMS bus (for example, use the LDIF Parser, XML Parser, and so forth). You can even use the Simple Object Access Protocol (SOAP) Parser to send SOAP requests over the JMS bus.

- If you don't have a Parser defined, the text body is returned in an attribute called message. When sending a message the Connector uses the provided message attribute to set the JMS text body part.

```
var str = work.getString ("message");
task.logmsg ("Received the following text: " + str );
```

If you expect to receive text messages in various formats (XML, LDIF, CSV ...) you must leave the Parser parameter blank and make the guess yourself as to what format the text message is. When you know the format you can use the `system.parseObject(`*`parserName`*`,` *`data`*`)` syntax to do the parsing for you:

```
var str = work.getString ("message");
// code to determine format
if ( isLDIF )
   e = system.parseObject( "ibmdi.LDIF", str );
else if ( isCSV )
   e = system.parseObject ( "ibmdi.CSV", str );
else
   e = system.parseObject ( "ibmdi.XML", str );
}
// Dump parsed entry to logfile
task.dumpEntry ( e );
```

The **Use Textmessage** flag determines whether the Connector must use this method when sending a message.

Object message

An object message is a message containing a serialized Java object. A serialized Java object is a Java object that has been converted into a byte stream in a specific format which makes it possible for the receiver to resurrect the object at the other end. Testing shows that this is fine as long as the Java class libraries are available to the JMS server in both ends. Typically, a java.lang.String object causes no problems but other Java objects might. For this reason, the JMS Connector does not generate object messages but is able to receive them. When you receive an object message the Connector returns two attributes:

java.object
: This attribute holds the java object and you must access the object using the getObject method in your **work**or **conn** entry.

java.objectClass
: This attribute is a convenience attribute and holds the class name (String) of the Java object

```
var obj = work.getObject ("java.object");
obj.anyMethodDefinedForTheObject ();
```

You only receive these messages.

Bytes message

A bytes message is a message carrying an arbitrary array of bytes. The JMS Connector generates this type of message when the **Use Textmessage** flag is **false**. The Connector takes the provided entry and serialize it into a byte array and send the message as a bytes message. When receiving a bytes message, the Connector first attempts to deserialize the byte array into an Entry object. If that fails, the byte array is returned in the message attribute. You must access the byte array using the getObject method in your **work** or **conn** entry.

```
var ba = work.getObject ("message");
for ( i = 0; i < ba.length; i++)
   task.logmsg ( "Next byte: " + ba [ i ] );
```

This type of message is generated only if**Use Textmessage** is **false** (not checked).

Iterator mode

A message selector is a String that contains an expression. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax. The message selector in the following example selects any message that has a NewsType property that is set to the value 'Sports' or 'Opinion':

```
NewsType = 'Sports' OR NewsType = 'Opinion'
```

Lookup mode

The Connector supports Lookup mode where the user can search for matching messages in a JMS Queue (Topic (Pub/Sub) is not supported by Lookup mode).

The Link Criteria specifies the JMS headers and properties for selecting matching messages on a queue.

For the advanced link criteria you must conform to the Message Selection specification as described in the JMS specification (http://java.sun.com/products/jms). The JMS Connector reuses the SQL filter specification (JMS message selection is a subset of SQL92) to build the message selection string. Turn on debug mode to view the generated message filter string.

There are basically two ways to perform a Lookup:

- Do a non-destructive search in a Queue (using **QueueBrowser**) which returns matching messages without removing the messages from the JMS queue.
- Removes all matching entries from the JMS queue.

Decide which to use by setting the **Lookup Removes** flag in the Connector configuration. For Topic connections the **Lookup Removes** flag does not apply as messages on topics are always removed when a subscriber receives it. However, the Lookup mode heeds the **Durable Subscriber** flag in which case the JMS server holds any messages sent on a topic when you are disconnected.

The JMS Connector works in the same way as other Connectors in that you can specify a maximum number of entries to return in your AssemblyLine settings. To ensure you retrieve a single message only during Lookup, specify **Max duplicate entries returned = 1** in the AssemblyLine settings. Setting **Max duplicate entries returned** to **1** enables you to retrieve one matching entry at a time regardless of the number of matching messages in the JMS queue.

Since the JMS bus is asynchronous the JMS Connector provides parameters to determine when the Lookup must stop looking for messages. There are two parameters that tells the Connector how many times it queries the JMS queue and for how long it waits for new messages during the query. Specifying **10** for the retry count and **1000** for the timeout causes the Connector to query the JMS queue ten times each waiting 1 second for new messages. If no messages are received during this interval the Connector returns. If during a query the Connector receives a message, it continues to check for additional messages (this time without any timeout) until the queue returns no more messages or until the received message count reaches the **Max duplicate entries returned** limit defined by the AssemblyLine. The effect of this is that a Lookup operation only retrieves those messages that are available at the moment.

AddOnly mode

In this mode, on each AssemblyLine iteration the JMS Connector sends an entry to the JMS server. If a Topic is used the message is published and if a Queue is used the message is queued.

Call/Reply mode

In this mode the Connector has two attribute maps, both **Input** and **Output**. When the AssemblyLine invokes the Connector, an Output map operation is performed, followed by an Input map operation. There is a method in the JMS Connector called `queryReply()` which uses the class QueueRequestor. The QueueRequestor constructor is given a non-transacted QueueSession and a destination Queue. It creates a TemporaryQueue for the responses and provides a `request()` method that sends the request message and waits for its reply.

JMS headers and properties

A JMS message consists of headers, properties and the body. Headers are accessed differently than properties and were not available in previous versions. In this version you can specify how to deal with headers and properties.

JMS headers

JMS headers are predefined named values that are present in all messages (although the value might be null). The following is a list of JMS header names this Connector supports:

JMSCorrelationID
(String) This header is set by the application for use by other applications.

JMSDeliveryMode
(Integer) This header is set by the JMS provider and denotes the delivery mode.

JMSExpires
(Long) A value of zero means that the message does not expire. Any other value denotes the expiration time for when the message is removed from the queue.

JMSMessageID

(String) The unique message ID. Note that this is not a required field and can be null.

Since the JMS provider might not use your provided message ID, the Connector sets a special property called **$jms.messageid** after sending a message. This is to insure that the message ID always is available to the user. To retrieve this value use `conn.getProperty("$jms.messageid")` in your **After Add** hook.

JMSPriority
(Integer) The priority of the message.

JMSTimestamp
(Long) The time the message was sent.

JMSType
(String) The type of message.

JMSReplyTo

(Destination) The queue/topic the sender expects replies to. When receiving a message this value holds the provider specific Destination interface object and is typically an internal Queue or Topic object. When sending a message you must either reuse the incoming Destination object or set the value to a valid topic/queue name. If the value is **NULL** (for example, an attribute with no values) or the string `"%this%"` the Connector uses its own queue/topic as the value. The difference between this method and explicitly setting the queue/topic name is that you need not update the attribute assignment if you change your Connector configuration's queue/topic name.

There is one restriction in the current version which enables you to only request a reply to the same type of connection as you are currently connected to. This means that you cannot publish a message on a topic and request the reply to a queue and vice versa.

It is not mandatory to respond to this header so the receiver of the message can completely ignore this field without any form of punishment.

These headers are all set by the provider and might be acted upon by the JMS driver for outgoing messages. In the configuration screen you can specify that you want all headers returned as attributes or specify a list of those of interest. All headers are named using a prefix of **jms.**. Also note that JMS header names always start with the string **JMS**. This means that you must never use property names starting with **jms.JMS** as they can be interpreted as headers.

JMS properties

In previous versions of this Connector all JMS properties were copied between the Entry object and the JMS Message. In this release you can refine this behavior by telling the Connector to return all user defined properties as attributes or specify a list of properties of interest. All properties are prefixed with **jms.** to separate them from other attributes. If you leave the list of properties blank and uncheck the **JMS Properties As Attributes** flag, you get the same behavior as for previous versions. Both JMS headers and JMS properties can be set by the user. If you use the backwards compatible mode you must set the entry properties in the **Before Add** hook as in:

```
conn.setProperty ( "jms.MyProperty", "Some Value" );
```

If you either check the **JMS Properties As Attributes** flag or specify a list of properties, you must provide the JMS properties as attributes. One way to do that is to add attributes using the **jms.** prefix in your attribute map. For example, if you add **jms.MyProperty** attribute map it results in a JMS property named **MyProperty**.

Configuration

The Connector name is JMS Pub/Sub Connector, and it needs the following parameters:

Broker
: The URL for the JMS server. When working with IPv6 addresses, this parameter must contain both the IPv6 JMS Server address as well as the JMS Server port. This parameter can also be used for providing the MQe initialization file.

  Note: The JMS Connector will support the IPv6 protocol if the JMS Server you connect to supports IPv6. IBM MQSeries® 5.3 does not.

Server Channel
: The name of the channel configured for the MQ server. This parameter only applies when the JMS Connector is used with IBM WebSphere MQ Server. This parameter is left in the configuration for backward compatibility

Use SSL Connection
: Enables the use of parameters and configuration settings required for SSL connection.

SSL Server Channel
: The name of channel configured for using SSL to access the MQ server. This parameter only applies when the JMS Connector is used with IBM WebSphere MQ Server. This parameter is left in the configuration for backward compatibility.

Queue Manager
: The name of Queue Manager defined for MQ server or INITIAL_CONTEXT_FACTORY for non-IBM MQ.

SSL CipherSuite
: Cipher Suite name which corresponds to cipher selected in configuring MQ server channel. This parameter only applies when the JMS Connector is used with IBM WebSphere MQ Server. This parameter is left in the configuration for backward compatibility.

User Name
: User name for authenticating access to the JMS.

Password
: Password for authenticating access to the JMS.

Connection Type
: Specify whether you are connecting to a **Queue** or **Topic** (Topic is sometimes called **Pub/Sub** for Publish/Subscribe).

Topic/Queue
: The topic/queue with which messages are exchanged.

Durable Topic Subscriber
Only relevant for **Connection Type Topic** (Pub/Sub). If **true**, this causes the Connector to create a durable subscriber. This means that the server stores messages for a topic for later retrieval when the Connector is offline.

Client ID
The client ID to use for Topic connections (mandatory for durable).

Message Selection Filter
Specifies a message filter for selection of messages from a Topic/Queue. Used in Iterator mode only.

GetNext Timeout
Time (in milliseconds) to wait for a new entry in Iterator mode. **-1** denotes **forever**.

JMS Server Type
Select the JMS server type. The full name of the class implementing the JMS Driver interface.

Specific Driver Attributes
These take the form of *name=value* driver attributes. For example:

```
QUEUE_FACTORY_NAME=primaryQCF, or
TOPIC_FACTORY_NAME=primaryTCF
```

JMS Driver Script
This parameter contains JavaScript code to be used for initialization of the JMS provider-specific objects. The contents of this parameter are passed to the configured JMS Driver using the "jsscript" Hashtable key name. This parameter is intended to be used by the JMS Script Driver, which executes the contents of this parameter as Javascript. This "jsscript" name is used as a key in the Hashtable passed to the JMS Script Driver. If the MQe or the MQ driver is configured to be used with the JMS Connector, then the contents of this parameter will be ignored. If a 3rd party JMS Driver different from the JMS Script Driver is configured the contents of this parameter will most likely be ignored.

For more details on the structure of this parameter's JavaScript code as well as on the environment in which it executes, please see the section labeled "JMS Script driver" in the section about the System Queue in the *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide* and the “System Queue Connector” on page 227.

Auto Acknowledge
If **true**, each message is automatically acknowledged by this Connector. If **false**, you must manually acknowledge the receipt of a JMS message (by means of the Connector's acknowledge() method). If **off**, use the JMS CLIENT_ACKNOWLEDGE mode.

Use Textmessage
If **true**, the Connector produces a Textmessage and sends the Entry object either by using the specified Parser to generate the text body or using the predefined message attribute as the text body.

JMS Headers as attributes
If **true**, all JMS headers are returned as attributes (prefixed by **jms.**) in Iterator and Lookup modes. For AddOnly mode, any attribute starting with **jms.JMS** is treated as JMS header. This causes these attributes to be set as JMS headers and removed from the Entry object before sending the message.

Note: only a few headers can be set, and setting them does not mean the JMS provider ever uses them.

Specific JMS Headers
Same as **JMS Headers as attributes**, but only the listed JMS headers are treated as headers. Specify one header per line.

JMS Properties as attributes
If **true**, all JMS properties are returned as attributes (prefixed by **jms.**) in Iterator and Lookup modes. For AddOnly mode, any attribute starting with **jms.** is treated as a JMS property. This causes these attributes to be set as JMS properties.

Specific JMS Properties
Same as **JMS Properties as attributes** , but only the listed JMS properties are treated as properties. Specify one property per line.

Lookup Removes
If **true**, each message found during Lookup is removed from the queue.

Note: You can set the **Max duplicate entries returned** parameter in your AssemblyLine Configuration settings to prevent Lookup from returning more than one entry.

If **false**, messages are returned as usual, but they are not removed from the queue.

Lookup Retries
The number of times Lookup searches the queue for matching messages.

Lookup Timeout
Time (in milliseconds) the Connector waits for new messages during a Lookup query. This parameter is used when **Lookup Removes** is set to **true** only.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

A Parser can be selected form the **Parser...** pane; once in this pane, choose a parser by clicking the bottom-right Inheritance button. If a Parser is specified, a JMS Text message is parsed using this Parser. This Parser works with messages that are received by the JMS Connector, and is used to generate a text message when JMS Connector sends a message.

Examples

Go to the *TDI_install_dir*/examples/SoniqMQ directory of your IBM Tivoli Directory Integrator installation.

Tivoli Directory Integrator 7.1 comes with an example of a JMS script driver for Sonic MQ. This sample demonstrates how the Tivoli Directory Integrator JMS components (JMS Connector, System Queue) can use the SonicMQ server as a JMS provider.

In directory *TDI_install_dir*/examples/was_jms_ScriptDriver you will find an example that demonstrates how to use the WebSphere Default JMS provider with the JMS Connector and the JMS Script Driver.

External System Configuration

The configuration of external JMS systems which this Connector accesses is not specific to this Connector. Any external JMS system which this Connector accesses must be configured as it would be configured for any other JMS client.

IBM WebSphere MQ

WebSphere MQ: When IBM WebSphere MQ is used as a JMS provider the following *jar* files have to be taken from the WebSphere MQ installation, and placed under the *TDI_install_dir*\jars\3rdparty\IBM folder:

For WebSphere MQ v6.0

- com.ibm.mqjms.jar (this will replace an existing file)
- com.ibm.mq.jar
- jms.jar

- connector.jar
- dhbcore.jar
- jta.jar

For WebSphere MQ v7.0

- com.ibm.mqjms.jar (this will replace an existing file)
- com.ibm.mq.jmqi.jar
- jms.jar
- dhbcore.jar

Enabling SSL:

The SSL (Secure Socket Layer) protocol enables secure communications with MQ queue managers. In order to enable it, adjustments must be made to the MQ server as well as the JMS Connectors in your IBM Tivoli Directory Integrator configuration. The steps below explain a sample setup.

Configuring SSL security for IBM WebSphere MQ v6.0 and v7.0

Managing certificates

To manage the SSL certificates on your local computer using a GUI, use IBM Key Management (iKeyman).

1. Create a key database file:

   Start iKeyman and select **Key database file -> New**. The "Key database type" must be **CMS**. You can choose the name and location of the file, but keep in mind that they must be set later in the queue manager's Key repository attribute. Check the **Stash the password to a file?** option and specify a password (it is used to access the file).

2. Obtain a certificate:

   You can request a certificate from a Certification Authority (CA), but for the purposes of this example we'll use a self-signed certificate. Select **Create -> New self-signed certificate** and complete the form. The "Key label" attribute value must be in the form <ibmwebspheremq<aQueueManagerNameinLowerCase>> (for examplem "ibmwebspheremqmyqueuemanager").

3. Extract the created certificate for further use:

   Use the **Extract certificate** button, specify a name, location and data type and click **OK**.

Configuring SSL on queue managers

For these configurations use WebSphere MQ Explorer.

1. Set the queue manager key repository:

   Select *your queue manager* **-> Properties -> SSL** and modify the value of the "Key repository" attribute. The value must be the location and name of the key database file from step 1 but without the .kdb extension.

Configuring SSL channels

1. Select *your queue manager* **-> Advanced -> Channels ->** *your channel name***.** Right-click and select **Properties -> SSL** and set a SSL CipherSpec (for this example set it to "NULL_MD5"). This specifies the encryption method and hash function used when sending the message.
2. Filtering certificates on their owner's name:

   Certificates contain the distinguished name of the owner of the certificate. You can optionally configure the channel to accept only certificates with attributes

in the distinguished name of the owner that match given values. To do this, select the **Accept only certificates with Distinguished Names matching these values** check box.

3. Authenticating parties initiating connections to a queue manager:

   When another party initiates an SSL-enabled connection to a queue manager, the queue manager must send its personal certificate to the initiating party as proof of identity. You can also optionally configure the queue manager's channel so that the queue manager refuses the connection if the initiating party does not send its own personal certificate. To do this, on the SSL page of the Channel properties dialog, select **Required** from the **Authentication of parties initiating connections** list. For this example, you won't need this additional check, so select **Optional**.

Configuring SSL security for the JMS connector

1. Additional settings for JMS Connector configuration:
   a. Check **Use SSL Connection**.
   b. Specify the "SSL Server Channel" which you configured in step Configuring SSL channels above.
   c. Specify the Queue Manager used.
   d. Select the **SSL_RSA_WITH_NULL_MD5** option from the **SSL CipherSuite** pull down list.
2. Adding the digital certificate to the IBM Tivoli Directory Integrator truststore:

   For this operation use iKeyman again.

   a. Adding the certificate:

      When an SSL connection is made, the queue manager will send its certificate as part of the initial handshake and the IBM Tivoli Directory Integrator truststore will be checked in order to validate the received certificate. If it is not validated the connection will be terminated.

      You can either edit the existing truststore file `testServer.jks` or create new Java keystore with the IBM Key Management tool. After that select the **Signer certificates** option from the combo box and click **Add**. Browse to the location you saved the extracted certificate from step 3 on page 159 and select it. When you are prompted for a label use the same as in point 2 on page 159 (ibmwebspheremqmyqueuemanager).

      If you chose **Required** for the **Authentication of parties initiating connections** option, you will need to create your own personal self-signed certificate in the IBM Tivoli Directory Integrator keystore and add it to the queue manager's key database file as signer certificate. The steps are identical with those specified above. This new certificate will be sent by our connector to the queue manager as part of the SSL handshake and if not present will result in termination of the connection.

      As stated above, this is not needed if you chose **Authentication of parties initiating connections -> Optional**.

   b. Modifying the `solution.properties` file:

      If you created new keystores or changed the location of the existing ones, this must be covered in `solution.properties`. For example:

```
javax.net.ssl.trustStore=C:\\Program Files\\IBM\\TDI\\V7.0\\jmsTrustStore
 javax.net.ssl.trustStorePassword=
 javax.net.ssl.trustStoreType=jks

 javax.net.ssl.keyStore=C:\\Program Files\\IBM\\WebSphere MQ\\Java\\bin\\jmsKeyStore
 javax.net.ssl.keyStorePassword=changeit
 javax.net.ssl.keyStoreType=jks
```

These modifications should be made prior to starting IBM Tivoli Directory Integrator.

Additional information:

If you uncheck the **Use SSL Connection** checkbox the following fields will be retained in the saved configuration, but not used in subsequent non-SSL connections:

1. SSL Server Channel
2. QueueManger
3. SSL CipherSuite

When the **Use SSL Connection** checkbox is NOT checked, the value specified for **Server Channel** will be used.

Considerations for Character encoding with IBM WebSphere MQ and the JMS Connector:

In case of multiple IBM WebSphere MQ servers residing on different platforms some problems related to mismatched character sets may occur. Here are some key points to consider when resolving such issues:

- The JMS Connector uses the IBM MQ implementation of the JMS API. Thus the character set conversion at the client side (that is, when doing MQGET) is enabled by default and there is no interface provided to change this behavior. The so-called data conversion appears when the message's character set is different than the destination's character set.
- If not explicitly set every Queue Manager has the default character set of the platform on which it resides. For example, on Linux – UTF-8, on z/OS – EBCDIC, and so forth.
- When putting messages in a Queue Manager make sure they are encoded in the same character set as the Queue Manager's character set. Doing so prevents putting messages encoded in one character set to Queue Manager expecting messages in another character set.
- Check whether your version of IBM WebSphere MQ supports conversion between the used character sets.

For a solution to a concrete scenario and workaround refer to section "Troubleshooting" on page 162.

For more information about data conversion in IBM WebSphere MQ you can use the following Web sources:

- http://www-01.ibm.com/support/docview.wss?uid=swg27005729&aid=1
- http://publibfp.boulder.ibm.com/epubs/pdf/csqzaw12.pdf
- http://publib.boulder.ibm.com/iseries/v5r2/ic2924/books/csqzae05.pdf
- http://publib.boulder.ibm.com/iseries/v5r2/ic2924/books/csqzak05.pdf
- http://www.elink.ibmlink.ibm.com/publications/servlet/pbi.wss?CTY=US&FNC=SRX&PBL=SC34658300

Lotus Expeditor Microbroker

If you have an existing Lotus Expeditor Microbroker installation, it can be used as a JMS provider. However there are different Microbroker versions or deployments; here are some example steps we have executed to use Microbroker:

1. Use the value `com.ibm.di.systemqueue.driver.IBMMB`for the parameter **JMS Server Type** (jms.driver) when working with Tivoli Directory Integrator. When configuring the JMS Password Store for the password plug-ins, you must use the value `com.ibm.di.plugin.pwstore.jms.driver.IBMMB` for the **JMS Server Type** (jms.driver) parameter.
2. Add the necessary .jar files to the ibmdisrv class path. Tivoli Directory Integrator was tested with the following version of Microbroker jars:

- com.ibm.micro.client.nl_3.0.0.1-20081111.jar
- com.ibm.micro.client_3.0.0.1-20081111.jar
- com.ibm.micro.utils.extended_3.0.0.1-20081111.jar
- com.ibm.micro.utils.nl_3.0.0.1-20081111.jar
- com.ibm.micro.utils_3.0.0.1-20081111.jar
- com.ibm.mqttclient.jms.nl_3.0.0.1-20081111.jar
- com.ibm.mqttclient.jms_3.0.0.1-20081111.jar
- com.ibm.mqttclient.nl_3.0.0.1-20081111.jar
- com.ibm.mqttclient_3.0.0.1-20081111.jar
- com.ibm.msg.client.osgi.jms_1.0.0.0.jar

  Note: You have to unpack the following jars file from this file:
  - com.ibm.msg.client.commonservices.jar
  - com.ibm.msg.client.jms.jar
  - com.ibm.msg.client.provider.jar
  - com.ibm.msg.client.jms.internal.jar
- com.ibm.msg.client.osgi.nls_1.0.0.0.jar

Notes:

a. If the Microbroker is installed on a different machine, the listed jar files have to be copied to a local folder different from *TDI_install_dir*/jars folder or any of its subfolders.

b. If some of the listed jars contain packed jars inside they also need to be unpacked and included in the Tivoli Directory Integrator Server classpath.

3. Add *TDI_install_dir*/jars/3rdparty/IBM/ibmjms.jar to the Tivoli Directory Integrator Server (ibmditk) classpath.

Here is an example of the modified ibmdisrv class path assuming that C:\MB_jars folder contains all the needed jars:

```
"%TDI_JAVA_PROGRAM%" -classpath "%TDI_HOME_DIR%\jars\3rdparty\IBM\ibmjms.jar;
C:\MB_jars\com.ibm.msg.client.osgi.jms_1.0.0.0.jar;C:\MB_jars\com.ibm.msg.client.osgi.nls_1.0.0.0.jar;
C:\MB_jars\com.ibm.msg.client.commonservices.jar;C:\MB_jars\com.ibm.msg.client.jms.jar;
C:\MB_jars\com.ibm.msg.client.provider.jar;C:\MB_jars\com.ibm.msg.client.jms.internal.jar;
C:\MB_jars\com.ibm.micro.client.nl_3.0.0.1-20081111.jar;C:\MB_jars\com.ibm.micro.client_3.0.0.1-20081111.jar;
fC:\MB_jars\com.ibm.micro.utils.extended_3.0.0.1-20081111.jar;C:\MB_jars\com.ibm.micro.utils.nl_3.0.0.1-20081111.jar;
C:\MB_jars\com.ibm.micro.utils_3.0.0.1-20081111.jar;C:\MB_jars\com.ibm.mqttclient.jms.nl_3.0.0.1-20081111.jar;
C:\MB_jars\com.ibm.mqttclient.jms_3.0.0.1-20081111.jar;C:\MB_jars\com.ibm.mqttclient.nl_3.0.0.1-20081111.jar;
C:\MB_jars\com.ibm.mqttclient_3.0.0.1-20081111.jar;
%TDI_HOME_DIR%\IDILoader.jar" %ENV_VARIABLES% com.ibm.di.loader.IDILoader com.ibm.di.server.RS %*
```

Note: The above list (which is one long line, broken up for visibility reasons) of .jar files might be different for the version of Microbroker you are using. Please consult the Microbroker documentation for the list of .jar files needed.

IBM WebSphere MQ Everyplace

When the bundled IBM WebSphere MQ Everyplace is used as a JMS provider, no additional *.jar* file copying is needed after Tivoli Directory Integrator is installed.

Troubleshooting

In case of systems containing two or more IBM WebSphere MQ servers exchanging messages, residing on different platforms the transmitted messages may be received corrupted.

For example consider the following scenario with two MQ servers – one MQ server on a z/OS platform sending messages to another MQ server on the Linux platform. If the received messages from the Linux MQ server are incorrect this might be due to a character set conversions since the default character set of the z/OS and Linux platforms are different. Here are some possible solutions when dealing with such an issue (in descending order, from most to least preferable):

1. Encode the messages using the z/OS MQ Queue Manager's character set before sending them to the z/OS MQ server
2. Configure the z/OS MQ Queue Manager to use the same character set as the expected messages
3. Use the following workaround with the correct z/OS and Linux character sets:
   a. Map the **message** attribute with an advanced mapping.
   b. Use the following script:
      ```
      ret.value = new java.lang.String(conn.getString("message").getBytes(z/OS_charset), Linux_charset);
      ```
   c. Run the configuration.

   Note: This workaround is only applicable for the described scenario. In systems with more than two MQ servers a more complex decoding of the messages may be needed.

JMS Password Store Connector

In previous releases, this Connector was known as the MQe Password Store Connector. In Tivoli Directory Integrator 7.1, it is called the JMS Password Store Connector; its name was changed because it is now able to make use of Tivoli Directory Integrator's JMS Driver pluggable architecture. This means that this connector can connect not only to the MQe Queue Manager but it can connect to the WebSphere MQ Queue Manager out of the box. In addition it can connect to any user provided Queue Manager as long as you provide the JMS Driver for establishing the connection.

The JMS Password Store Connector supports Iterator mode only.

The JMS Password Store can use PKCS7 encryption to sign and encrypt the password change notification messages before it sends them to the JMS Password Store Connector.

Notes:

1. This component is not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.
2. For more information about installing and configuring the IBM Password Synchronization plug-ins, please see the *IBM Tivoli Directory Integrator V7.1 Password Synchronization Plug-ins Guide*.
3. IBM Tivoli Directory Integrator 7.1 components can be deployed to take advantage of MQe Mini-Certificate authenticated access. To use these MQe features, it is necessary to download and install IBM WebSphere MQ Everyplace® 2.0.1.7 (or higher) and IBM WebSphere MQ Everyplace Server Support ES06. Use of certificate authenticated access prevents an anonymous MQe client Queue Manager or applications submitting a change password request to the JMS Password Store Connector.
4. IBM MQ Everyplace does not support IP Version 6 addressing; as a consequence, the JMS Password Store Connector can only reach MQe using traditional IPv4 addresses.
5. IBM MQ Everyplace is deprecated in this version of Tivoli Directory Integrator, and will be removed in a future version. A suitable lightweight message queue will be provided at that time.

The JMS Password Store Connector supports receiving messages from multiple password stores.

Connector Workflow

The following is the JMS Password Store Connector workflow:

1. The JMS Password Store Connector requests a message from a predefined queue on either its local MQe Queue Manager (if using the MQe JMS Driver) or the external Queue Manager (if using any other than the MQe JMS Driver). The messages are retrieved using the JMS interface.
2. The retrieved message is verified and/or decrypted (this step is optional).
3. The message is parsed and an Entry object is created. The attributes of this Entry object represent the user ID, the password values and the type of password update.
4. This newly created Entry object is passed to the IBM Tivoli Directory Integrator AssemblyLine.

On initialization, the JMS Password Store Connector performs the following actions:

Using the MQe JMS Driver:

- The Connector starts the MQe Queue Manager if it is not already started and gets a reference to the running Queue Manager.
- Initiates a connection to the Storage Component and notifies the Storage Component that the JMS Password Store Connector Queue Manager is up and ready for receiving notifications.

Using other than the MQe JMS Driver:

The specific JMS Driver is initialized and a connection is established with the external Queue Manager.

On getting a password update message, the Connector can operate in one of three modes:

No wait
Checks if password update message is available in the QueueManager queue. If **yes**, the mode retrieves and parses the message. If **no**, the mode returns **NULL**, signalling the end of the Iterator.

Number of milliseconds to wait
Waits for a specified number of milliseconds for a password update message to appear in the QueueManager queue. If the password update message appears, this mode retrieves and parses the message. If not, this mode returns **NULL**, signaling the end of the Iterator.

Wait forever
The Connector waits until a password update message appears in the QueueManager queue. It never returns **NULL**, and when operating in this mode it must be stopped externally.

By default, the Connector automatically acknowledges every message it receives from the QueueManager JMS queue. However, you can change this behavior by de-selecting the **Auto Acknowledge** parameter; in that case, you are responsible for message acknowledgements yourself by calling the Connector's `acknowledge()` method at appropriate places in the AssemblyLine. Each time you call the Connector's `acknowledge()` method you acknowledge all messages delivered so far by the Connector.

Force transfer of accumulated messages from the JMS Password Store with MQe

Accumulated messages in an MQe-based Password Store are not automatically transferred to the Tivoli Directory Integrator. To force transmission of such accumulated messages, use the **Storage notification server(s)** parameter of the JMS Password Store Connector and the "mqe.notify.port" parameter of the JMS Password Store.

Here is an explanation:

When the JMS Password Store is used with MQe, there are two MQe Queue Managers involved – one on the Password Store side and the other on the Tivoli Directory Integrator side. On the Password Store side a remote MQe queue is configured, which points to a local MQe queue on the Tivoli Directory Integrator side.

Messages are transferred only when both Queue Managers are operational. When Tivoli Directory Integrator is not running, the Queue Manager of the Password Store accumulates arriving messages. Normally MQe does not automatically detect when a remote Queue Manager goes operational. So when the Directory Integrator goes back online, the accumulated messages are not transferred until a new message arrives in the Password Store.

There is a special feature which allows the JMS Password Store Connector to "pull" accumulated messages from the Password Store. This feature is configured by the **Storage notification server(s)** parameter of the Connector and the "mqe.notify.port" parameter of the JMS Password Store. When the Connector initializes, it sends a notification to the Password Store to start sending accumulated messages. Note that currently there is no "push" alternative, that is, the Password Store does not periodically check if Tivoli Directory Integrator is running.

PKCS7 Encryption support

The JMS Password Store can use PKCS7 encryption to sign and encrypt the password change notification messages before it sends them to the JMS Password Store Connector. The use of PKCS7 encapsulation is optional; by default it is turned off. Both signing and encryption need certificates in order to function. Usage of PKCS7 is incompatible with the older PKI-based encryption mechanisms available in older versions of Tivoli Directory Integrator.

With the PKCS7 option activated, it verifies the signature of each received message by comparing the Signer certificate with those in its trust store. In case of a match it verifies the message signature. If the signature verification is successful the Connector accepts the message and decrypts it with the Connector's private key from its own certificate.

Note: If PKCS7 needs to be used then both the JMS Password Store Connector and the JMS Password Store (all of them, if multiple Stores are used) need to be setup to use PKCS7. If only one side is configured to use PKCS7 then an error will occur.

The certificates are stored in a .jks file. The Connector has a .jks file and the JMS Password Store has another .jks file.

Signing of messages

Signing is used to verify that the sender of the message is the one he/she claims to be.

In this particular scenario the JMS Password Store Connector needs to verify that the sender of a password change notification message is actually a trusted JMS Password Store.

It is possible to have several password stores sending messages to a single JMS Password Store Connector. In this case the Connector must be configured so that its .jks file contains the public keys of each of the trusted password stores.

Encryption of messages

Encryption is achieved by having the password store use the public key of the Connector to encrypt the message. Then the Connector uses its private key to decrypt the message.

Certificate management

A .jks file is required in order to be able to work with the PKCS7 functionality. It must contain not only the JMS Password Store Connector's certificate, but also the certificates of all the password stores that send messages to it.

The JMS Password Store Connector's certificate is a self-signed personal certificate, whose private key is used to decrypt the messages from the password store.

The password stores' certificates are trusted signer certificates, which are supplied from each JMS Password Store's .jks file. Every received message is then verified: the public key, attached to it, is compared with the available in the .jks file. In case of a match the message signature is verified against the certificate and then the message is decrypted using the Connector's own private key.

Certificate structure: Certificates are stored in a .jks file. The Connector has a .jks file and the password store has another, corresponding, .jks file. The two .jks files need to contain the following so that PKCS7 can be used:

JMS Password Store .jks file

- The public key of the Connector as a trusted signer certificate
- The private-public key pair of the password store

JMS Password Store Connector .jks file

- The public key of each trusted password store as a trusted signer certificate
- The private-public key pair of the Connector

Creating certificates: The primary tool used to handle .jks files is `ikeyman.exe`. Ikeyman.exe is a tool available with every JVM distributed with Tivoli Directory Integrator.

It can be found in: *`TDI_install_dir`*`\jvm\jre\bin`, where *TDI_install_dir* is the installed directory of IBM Tivoli Directory Integrator. Below are the steps you can follow in order to create the required keystore/truststore .jks files.

1. **Creating a .jks file**

   To create a new .jks file click on **Key Database File -> New** and choose JKS together with the desired name and file path. You will be asked to enter a password. Remember it – it has to be provided later when setting up the components. You will need to create at least two such files – one for the MQePasswordStore and another one for the JMS Password Store Connector.

2. **Creating a certificate**

   To create a new certificate click on the drop-down menu above the list of certificates and choose **Personal Certificates**. Next, click on **New Self-Signed...** and enter the appropriate information.

3. **Transferring certificates**

   The last step is adding the just created self-signed certificates from the MQePasswordStore's JKS to the JMS Password Store Connector's and vice versa. For this purpose you have to extract the certificate as DER binary data: click on **Extract Certificate...** and then choose **Data Type -> DER Binary data**. Save it to an appropriate location with the desired name and open the other .jks file. Click **Add...** and find the file with the DER extracted data (Note: you must have chosen the **Signer Certificates** list before adding the new certificate).

Note: The implementation of PKCS7 in Tivoli Directory Integrator 7.1 does not support certificates that are secured with an additional password except the one set for the .jks file.

Example usage

The following example demonstrates how the JMS Password Store Connector can be configured to work with the configured JMS Password Store, described in *IBM Tivoli Directory Integrator V7.1 Password Synchronization Plug-ins Guide*. Parameter **PKCS7** is checked – meaning that the PKCS7 encryption/certification option is enabled.

The path to the .jks file, parameter **PKCS7 Key Store File** is `C:\dev\di611_061025a\certs\mqeconnpkcs7.jks`. It must contain its self-signed certificate as well as the trusted signer certificate of the JMS Password Store (please refer to “Creating certificates” on page 167 for more information about creating the necessary certificates). In our case the parameter **MQeConnector Certificate Alias** is specified as "*mqeconn*".

For the needs of our example we need to create the two .jks files – '`mqepkcs7.jks`' and '`mqeconnpkcs7.jks`'. The steps are as follows:

1. Open iKeyman.exe and click on **Key Database File-> New...**
2. Select the desired location of the file. For the example described above, save the .jks file under `C:\dev\061025a\certs` with the name `mqeconnpkcs7.jks`. By pressing the **OK** button, you will be asked to enter a password. To keep compatibility with the other data in the example, enter "*secret*" as password.
3. The next step is to create the JMS Password Store Connector's certificate itself. For this purpose select *Personal Certificates* from the drop-down menu and click **New Self-Signed...** The Key Label is the alias of the certificate in the .jks file. Set it to "*mqeconn*". The other options can be left with the default values.
4. Extract the just created self-signed certificate "mqeconn" as DER data in the same folder: `C:\dev\di611_061025a\certs`. Choose a name that corresponds to the certificate itself (for example, mqeconn). This file will be used later to import the JMS Password Store Connector's certificate in the .jks file of a JMS Password Store.
5. Repeat the steps from 1 to 4, but this time the location of the .jks file is: `C:\Program Files\IBM\DiPlugins\mqepkcs7.jks` and the password again: "*secret*". For Key Label of the JMS Password Store certificate set the value to "*mqestore*" and extract it as `mqestore.der` in the same directory: `C:\Program Files\IBM\DiPlugins\`.
6. Both created .jks files must exchange their certificates. Since the `mqepkcs7.jks` file is opened, import first the DER binary data that was extracted from `mqeconnpkcs7.jks`. Select *Signer Certificates* from the drop-down list and click on **Add...** In the window that popped up select "*Binary DER data*" as Data

type and then browse to the location `C:\dev\di611_061025a\certs`, where the .der file is saved. Select the `mqeconn.der` file and click "**OK**". A label for the imported certificate is required. To avoid confusion it is advisable to give it the same alias as in the other .jks file, in this case, mqeconn, because this value must be given in the properties file of the JMS Password Store for the property "pkcs7MqeConnectorCertificateAlias".

7. The same procedure must be performed on the `mqeconnpkcs7.jks` file (the key store holding the necessary certificates for the connector). First open the .jks file by clicking **Key Database File-> Open...** and navigating to the exact location. If you followed all the instructions precisely the path to the required file should be `C:\dev\di611_061025a\certs`. The password will be prompted for again. Afterwards repeat step 6 on page 168 with the new parameters. The location is `C:\Program Files\IBM\DiPlugins` and the certificate name is `mqestore.der`. For convenience name it "*mqestore*" again. With this step the example is completed.

Schema

The JMS Password Store Connector constructs IBM Tivoli Directory Integrator Entry objects with the following fixed attribute structure (schema):

UserID
Contains a single string value.

UpdateTypes
Contains one of the following string values:

- **replace** (replace password values operation)
- **add** (add password values operation)
- **delete** (delete password values operation)

Passwords
A multi-valued attribute. Each value is a string representing a password value.

Configuration

GetNext Timeout
Specify the number of milliseconds the Connector waits for a new password update message to appear in the QueueManager queue. Specify **-1** to wait forever, and **0** to return immediately if no message is available (returning NULL).

Storage notification server(s)
Specify in a *host*:*port* format the Storage Component server that listens for notifications from the JMS Password Store Connector. The default value for the port is **41002** and the host must be the IP address of the machine where the Password Synchronizer and the Storage Component are deployed.

There can be multiple Storage Component servers; specify each on a separate line.

Broker
The URL for the JMS server. When working with IPv6 addresses, this parameter must contain both the IPv6 JMS Server address as well as the JMS Server port. This parameter can also be used for providing the MQe initialization file.

JMS Server Type
The full name of the class implementing the JMS Driver interface; you can select one of the following values:

- IBMMQ
- IBMMQe

Specific Driver Attributes
These take the form of *name=value* driver attributes. For example:

`QUEUE_FACTORY_NAME=primaryQCF`, or
`TOPIC_FACTORY_NAME=primaryTCF`

Server Channel
The name of the channel configured for the MQ server. This parameter only applies when the JMS Password Connector is used with IBM WebSphere MQ Server. This parameter is left in the configuration for backward compatibility

Queue Manager
The name of Queue Manager defined for MQ server or INITIAL_CONTEXT_FACTORY for non-IBM MQ.

User Name
The User name for authenticating access to the JMS.

Password
The Password for authenticating access to the JMS.

Client ID
The client ID to use for Queue connections.

Use SSL Connection
Enables the use of parameters and configuration settings required for SSL connection.

SSL Server Channel
The name of channel configured for using SSL to access the MQ server. This parameter only applies when the JMS Connector is used with IBM WebSphere MQ Server. This parameter is left in the configuration for backward compatibility.

SSL CipherSuite
The Cipher Suite name which corresponds to the cipher selected in configuring MQ server channel. This parameter only applies when the Connector is used with IBM WebSphere MQ Server. This parameter is left in the configuration for backward compatibility.

Auto Acknowledge
If checked each message is automatically acknowledged, otherwise messages should be acknowledged manually through the Connector's `acknowledge()` method. Default is selected.

Decrypt messages
Check this field if the Storage Component encrypts the password update messages and they need to be decrypted by the Connector.

Key Store File
The path of the JKS file used to decrypt password data (only taken into account when the Decrypt messages field is selected).

Key Store File Password
The password of the JKS file (only taken into account when the Decrypt messages field is selected).

Key Store Certificate Alias
The alias of the key from JKS file (only taken into account when the Decrypt messages field is selected).

Key Store Certificate Password
The password used to retrieve the private key. If not specified, the **Key Store File Password** is used to retrieve the private key (only taken into account when the Decrypt messages field is selected).

PKCS7
This indicates whether PKCS7 encapsulation is used or not. The default value is disabled. All other parameters related to the PKCS7 functionality are considered if only this parameter is enabled.

PKCS7 Key Store File
This is the full path to the JMS Password Store Connector's JKS together with its name. There is no need for double slash "\\" instead of single "\", when specifying the file path on Windows platforms.

PKCS7 Key Store File Password
The actual, plaintext value of the password for the JKS file (whereas for the MQePasswordStore's property the encrypted version is required).

MQeConnector Certificate Alias
The alias of the JMS Password Store Connector's certificate as it is saved in the .jks file without any extensions.

Detailed Log
Check this field for more detailed log messages.

JMS drivers

IBM WebSphere MQe driver

In order to use MQe as the JMS provider for the JMS Password Store Connector, the **JMS Server Type** config parameter must be set to "IBMMQE", and the "systemqueue.jmsdriver.name" property in `global.properties` or `solution.properties` must be set to "com.ibm.di.systemqueue.driver.IBMMQe".

The IBM WebSphere MQe driver has one parameter:

- **mqe.file.ini** - the value of this parameter must be the absolute filename of the MQe initialization file.

For example, if the JMS Password Store Connector needs to be configured to use MQe, then the following line must be put in `global.properties` or `solution.properties`:

```
systemqueue.jmsdriver.param.mqe.file.ini=TDI_install_folder/MQePWStore/pwstore_server.ini
```

This is the default location where the MQe Configuration utility creates the MQe initialization file.

Note: In order to be able to use MQe as the JMS provider for the JMS Password Store Connector an MQe Queue Manager needs to be created. This can be done using the MQe Configuration utility bundled with Tivoli Directory Integrator; for more information, see "MQe Configuration Utility" in *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*.

IBM WebSphere MQ driver

In order to use IBM WebSphere MQ as the JMS provider for the JMS Password Store Connector the **JMS Server Type** config parameter must be set to "IBMMQ".

The IBM WebSphere MQ driver has the following parameters:

- **Broker** (jms.broker) - the MQ server address (IP address and TCP port number); an example value would be "192.168.113.54:1414"
- **Server Channel** (jms.serverChannel) - the name of the server channel configured for the MQ server instance.
- **Queue Manager** (jms.qManager) - the name of the Queue Manager defined for the MQ server instance.
- **SSL Cipher Suite** (jms.sslCipher) - the cipher suite name which corresponds to the cipher selected when configuring the MQ server channel; an example value would be "SSL_RSA_WITH_RC4128_MD5".
- **Use SSL Connection** (jms.sslUseFlag) - specifies whether SSL will be used on the connection to the MQ Server instance; valid values are **true** and **false**.

For specific configuration of the Websphere MQ server, please refer to its documentation.

See also

"System Queue Connector" on page 227,
The chapter on System Queue in the *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*,
JMS Password Store in *IBM Tivoli Directory Integrator V7.1 Password Synchronization Plug-ins Guide*,
"JMS Connector" on page 151.

JMX Connector

The JMX Connector uses the JMX 1.2 and JMX Remote API 1.0 specifications. It only uses standard JMX features.

The JMX Connector can listen to, and report, either local or remote JMX notifications, depending on how it is configured.

When the AssemblyLine starts the JMX Connector is initialized. On initialization, the Connector determines whether it will report local or remote notifications based on the Connector parameters (the Connector cannot report both local and remote notifications in a single run). Then, the Connector gets either a local or a remote reference to the respective MBean Server and registers for the desired JMX notifications specified in a Connector parameter.

In the `getNextEntry()` method, the Connector blocks the AssemblyLine while waiting for notifications. When a notification is received, the `getNextEntry()` method of the Connector returns an Entry (which contains the notification details) to the AssemblyLine.

Notifications that are received between successive `getNextEntry()` calls are buffered, so that no notifications are lost. If there are buffered notifications when the `getNextEntry()` is called, then the Connector returns the first buffered notification immediately without blocking the AssemblyLine.

This Connector operates in Iterator mode only.

Connector Schema

The JMX Connector makes the following Attributes available (Input Attribute Map):

event.originator
The JMX Connector object of type com.ibm.di.connector.JMXConnector

event.type
The notification type of type java.lang.String

event.rawNotification
The raw JMX Notification instance received by the JMX Connector (javax.management.Notification). If the component that broadcasts this notification has extended javax.management.Notification and has put some additional data in the subclass, this extra information can be retrieved through this property.

event.timestamp
The notification timestamp of type java.lang.Long. It represents the moment when the notification was created.

event.sequenceNumber
The notification sequence number (java.lang.Long). It represents the notification sequence number within the source object. It's a serial number identifying a particular instance of notification in the context of the notification source. The notification model does not assume that notifications will be received in the same order that they are sent. The sequence number can be used to sort received notifications.

event.message
The message of the notification (java.lang.String).

event.mbean.objectName
The object name of the registered and unregistered MBean (javax.management.ObjectName). This property is only available if the event.type is JMX.mbean.registered or JMX.mbean.unregistered. ObjectName represents an MBean Name (as well as a wildcard for MBean Names). The entire combination of the domain plus all keys and values must be unique. (That is equivalent to saying that the entire MBean Name must be unique).

event.mbean.name
The string representation of the MBean object name (java.lang.String). This property is only available if the event.type is JMX.mbean.registered or JMX.mbean.unregistered.

event.userData
The JMX notification user data (java.lang.Object).

event.source
The MBean object name on which the notification initially occurred (javax.management.ObjectName).

Configuration

Mode This parameter determines whether the JMX Connector will listen for local or remote JMX notifications. The Connector registers for and listens to remote JMX notifications according to the JMX Remote API 1.0 specification.

The available values (drop-down list) for this parameter are *remote* and *local*.

The value "local" means that the Connector will only listen for notifications issued by MBeans registered with an MBeanServer in the local Java Virtual Machine.

The value "remote" means that the Connector will connect to a remote JMX system based on the JMX Remote API 1.0 specification, and register for notifications issued by MBeans registered with an MBean server in the Java Virtual Machine of that remote system.

Remote JMX URL
This parameter is only taken into account if the "mode" parameter is set to "remote". This is the JMX URL used to connect to the remote JMX system. More precisely, this URL is specified by the remote MBean Server on its startup and is used by remote clients to connect to it.

An example value for this parameter would be: "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxconnector"

The default value is "service:jmx:rmi://localhost/jndi/jmx"

Listen to all MBeans
Specifies whether the Connector will register with all available MBeans (checked) or only with the ones specified in the **MBeans to listen to** Connector parameter (unchecked). This parameter is checked by default.

MBeans to listen to
Specifies a list of MBean object names, each typed on a separate line. This list specifies the MBeans with which the Connector will register for notifications. If no MBean object names are specified (that is, the list is empty) notifications issued by any MBean will be reported. If at least one MBean name is specified, then only notifications issued from the MBeans specified will be reported.

Notification types
The type of a JMX notification, not to be confused with its Java class, is the characterization of a generic notification object. The type is assigned by the broadcaster object and conveys the semantic meaning of a particular notification. The type is given as a String field of the Notification object. This string is interpreted as any number of dot-separated components, allowing an arbitrary, user-defined structure in the naming of notification types.

Specifies the types of JMX notifications which the JMX Connector will listen to. Notifications whose types are not specified will not be reported by the Connector. Each notification type must be typed on a separate line.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

The JMX Connector is capable of using the SSL protocol on the connection. If the remote JMX system accepts only SSL connections, the JMX Connector will automatically establish an SSL connection provided that a trust store is configured properly. This means that appropriate values have to be set for the `javax.net.ssl.trustStore`, `javax.net.ssl.trustStorePassword` and `javax.net.ssl.trustStoreType` properties in `global.properties` or `solution.properties`.

See also

Wikipedia on JMX,
Getting Started with JMX,
JMX Tutorial,
Managing TDI with ITM using JMX.

JNDI Connector

The JNDI Connector provides access to a variety of JNDI services; it uses the *javax.naming* and *javax.naming.directory* packages to work with different directory services. To reach a specific system, you must install the JNDI driver for that system, for example `com.sun.jndi.ldap.LdapCtxFactory` for LDAP. The driver is typically distributed as one or more jar or zip files. Place these file in a place where the Java runtime can reach them, for example, in the *TDI_install_dir*`/lib/ext` directory.

This Connector supports Delta Tagging at the Attribute level. This means that provided a previous Connector in the AssemblyLine has provided Delta information at the Attribute level, the JNDI Connector will be able to use it in order to make the changes needed in the target JNDI directory.

When using the JNDI Connector for querying an LDAP Server, a *SizeLimitExceededException* may occur if the number of entries satisfying the search criteria is greater than the maximum limit set by the LDAP Server. To work around this situation, either increase the LDAP Server's maximum result limit, or set the `java.naming.batchsize` provider parameter to some value smaller than the maximum limit of the server. For more information on the java.naming.batchsize parameter refer to: http://java.sun.com/products/jndi/tutorial/ldap/search/batch.html

Configuration

The Connector needs the following parameters:

JNDI Driver
The class name (the JNDI Naming factory) for the JNDI driver.

Provider URL
The URL for the connection, for example, ldap://host for the LDAP driver.

Authentication Method
Type of LDAP authentication. Can be one of the following:

- **Anonymous** - If this authentication method is set then the server, to which a client is connected, does not know or care who the client is. The server allows such clients to access data configured for non-authenticated users. The Connector automatically specifies this authentication method if no username is supplied. However, if this type of authentication is chosen and **Login username** and **Login password** are supplied, then the Connector automatically sets the authentication method to Simple.
- **Simple** - using **Login username** and **Login password**. Treated as anonymous if **Login username** and **Login password** are not provided. Note that the Connector sends the fully qualified distinguished name and the client password in cleartext, unless you configure the Connector to communicate with the LDAP Server using the SSL protocol.
- **CRAM-MD5** - This is one of the SASL authentication mechanisms. On connection, the LDAP Server sends some data to the LDAP client (that is, this Connector). Then the client sends an encrypted response, with password, using MD5 encryption. After that, the LDAP Server checks the password of the client. CRAM-MD5 is supported only by LDAP v3 servers. It is not supported against any supported versions of Tivoli Directory Server.
- **SASL** - The client (this Connector) will use a Simple Authentication and Security Layer (SASL) authentication method when connecting to the LDAP Server. Operational parameters for this type of authentication will need to be specified using the **Extra Provider Parameters** option; for example, in order to setup a DIGEST-MD5 authentication you will need to add the following parameter in the Extra Provider Parameters field:

  ```
  java.naming.security.authentication:DIGEST-MD5
  ```

  For more information on SASL authentication and parameters see: http://java.sun.com/products/jndi/tutorial/ldap/security/sasl.html.

Note: Not all directory servers support all SASL mechanisms and in some cases do not have them enabled by default. Check the documentation and configuration options for the directory server you are connecting to for this information.

Login username
The principal name (for example, **username**).

Login password
The credentials (for example, **password**).

Use SSL
Uses secure sockets layer for communication with LDAP server.

Name parameter
Specify which parameter in the AssemblyLine entry is used for naming the entry. This is used during add, modify and delete operations and returned during read or search operations. If not specified, **$dn** is used.

Search Base
The search base used when iterating the directory. Specify a distinguished name. Some directories enable you to specify a blank string which defaults to whatever the server is configured to do. Other directory services require this to be a valid distinguished name in the directory.

Search Filter
The search filter to be used when iterating the directory.

Search Scope
The search scope to be used when iterating the data source. Possible values are:

subtree
Return entries on all levels from search base and below.

onelevel
Only return entries that are immediately below searchbase.

Referrals
Specifies how referrals encountered by the LDAP server are to be processed. The possible values are:

- **follow** – Follow referrals automatically.
- **ignore** – Ignore referrals.
- **throw** – Throw a ReferralException when a referral is encountered. You need to handle this in an error Hook.

Extra Provider Parameters
A list of extra provider parameters you want to pass to the provider. Specify each *parameter*:*value* on a separate line. For example:

```
java.naming.batchsize=100
```

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Setting the Modify operation

The JNDI connector has a way to set a **modify operation** value when the connector is in Modify mode. You can also use the simple connector interface to directly add, remove or replace attribute values and attributes instead of setting **modify operation**.

There is no Config Editor provided to set the **modify operation**. You must manually add the operation value to each attribute in the work entry of the JNDI connector in Modify mode using the following interface:

di.com.ibm.di.entry.Attribute.setOper(char operation) operation

di.com.ibm.di.entry.Attribute.ATTRIBUTE_DELETE
This constant deletes the specified attribute values from the attribute.

The resulting attribute has the set difference of its prior value set and the specified value set. If no values are specified, it deletes the entire attribute. If the attribute does not exist, or if some or all members of the specified value set do not exist, this absence might be ignored and the operation succeeds, or an Exception might be thrown to indicate the absence. Removal of the last value might remove the attribute if the attribute is required to have at least one value.

di.com.ibm.di.entry.Attribute.ATTRIBUTE_REPLACE
This constant replaces an attribute with specified values.

If the attribute already exists, this constant replaces all existing values with new specified values. If the attribute does not exist, this constant creates it. If no value is specified, this constant deletes all the values of the attribute. Removal of the last value might remove the attribute if the attribute is required to have at least one value. This is the default modify operation.

di.com.ibm.di.entry.Attribute.ATTRIBUTE_ADD
This constant adds an attribute with the specified values.

If the attribute does not exist, this constant creates the attribute. The resulting attribute has a union of the specified value set and the prior value set.

Calling the Modify Interface

Adding a value to an attribute:

```
public void addAttributeValue(String moddn, String modattr, String modval)
```

throws Exception where:

- *moddn* is the DN to which you want to add the attribute value
- *modattr* is the name of the attribute to which you want to add a value
- *modval* is the value you want to add to *modattr*

For example, if you want to add `"cn=bob"` to the **members** attribute of `"cn=mygroup"` you use the method as such:

```
thisConnector.connector.addAttributeValue("cn=mygroup","members","cn=bob");
```

An Exception is thrown when the underlying modify operation fails.

Replacing the attribute value:

```
public void replaceAttributeValue(String moddn, String modattr, String modval)
```

throws Exception where:

- *moddn* is the DN to which you want to add the attribute value
- *modattr* is the name of the attribute to which you wish to add a value
- *modval* is the value you want to add to *modattr*

For example, if you want to replace the **members** attribute of "cn=mygroup" with "cn=bob" only, you use the method as such:

```
thisConnector.connector.replaceAttributeValue("cn=mygroup","members","cn=bob");
```

An Exception is thrown when the underlying modify operation fails.

Removing attribute:

```
public void removeAttribute(String moddn, String modattr)
```

throws Exception where:

- *moddn* is the DN from which you want to remove all attribute values
- *modattr* is the attribute name for which you want to remove all values

For example, if you want to remove the **members** attribute of "cn=mygroup" you use the method as such:

```
thisConnector.connector.removeAttribute("cn=mygroup","members");
```

An Exception is thrown when the underlying modify operation fails.

Removing a certain attribute value from an attribute:

```
public void removeAttributeValue(String moddn, String modattr, String modval)
```

throws Exception where:

- *moddn* is the DN from which you want to remove the attribute value
- *modattr* is the attribute name that you want to change
- *modval* is the value you want to remove from given attribute

An Exception is thrown when the underlying modify operation fails.

modify operation

modify operation can be set per Modify request. It causes **modify operation** for all attributes in the modify request entry to be set to the proper modify operation value. Property values and matching modify operation values:

Property value (String)	modify operation value
delete	di.com.ibm.di.entry.Attribute. ATTRIBUTE_DELETE
add	di.com.ibm.di.entry.Attribute. ATTRIBUTE_ADD
replace	di.com.ibm.di.entry.Attribute. ATTRIBUTE_REPLACE

This property can be set at any time while the Connector is running by setting the property **modOperation** from the scripts:

```
conn.setProperty("modOperation","delete");
```

Note: This property does not affect the behavior of the any interfaces defined above. However, it does overwrite the existing **modify operation** set by `di.com.ibm.di.entry.Attribute.setOper(char operation)`

Skip Lookup in Update and Delete mode

The JNDI Connector supports the **Skip Lookup** general option in Update or Delete mode. When it is selected, no search is performed prior to actual update and delete operations. It requires a name parameter (for example, $dn for LDAP) to be specified in order to operate properly.

See also

JNDI overview,
JNDI Tutorial,
JNDI FAQ,
"LDAP Connector" on page 181.

LDAP Connector

The LDAP Connector provides access to a variety of LDAP-based systems. The Connector supports both LDAP version 2 and 3. It is built layered on top of JNDI connectivity.

This Connector can be used in conjunction with the IBM Password Synchronization plug-ins. For more information about installing and configuring the IBM Password Synchronization plug-ins, please see the *IBM Tivoli Directory Integrator V7.1 Password Synchronization Plug-ins Guide*.

Note that, unlike most Connectors, while inserting an object into an LDAP directory, you must specify the object class attribute, the **$dn** attribute as well as other attributes. The following code example, if inserted in the Prolog, defines an **objectClass** attribute that you can use later.

```
// This variable used to set the object class attribute
var objectClass = system.newAttribute ("objectclass");
objectClass.addValue ("top");
objectClass.addValue ("person");
objectClass.addValue ("inetorgperson");
objectClass.addValue ("organizationalPerson");
```

Then your LDAP Connectors can have an attribute called **objectclass** with the following assignment:

```
ret.value = objectClass
```

To see what kind of attributes the **person** class has, see http://java.sun.com/products/jndi/tutorial/ldap/schema/object.html

You see that you must supply an **sn** and **cn** attribute in your Update or Add Connector.

In the LDAP Connector, you also need the **$dn** attribute that corresponds to the distinguished name. When building **$dn** in the Attribute Map, assuming an attribute in the work object called **iuid**, you typically have code like the following fragment:

```
var tuid = work.getString("iuid");
ret.value = "uid= " + tuid + ",ou=people,o=example_name.com";
```

Notes:

1. The two special attributes, **$dn** and **objectclass** usually are not included in Modification in Update mode unless you want to move entries in addition to updating them.
2. If you cannot connect to your directory, make sure the **Use SSL** flag in the Configuration is set according to what the directory expects.
3. When doing a Lookup, you can use **$dn** as the Connector attribute, to look up using the distinguished name. Do not specify a Simple Link Criteria using both **$dn** and other attributes; in this case a simple lookup will be done with the DN using an Equals comparison.
4. Certain servers have a size limit parameter to stop you from selecting all their data. This can be a nuisance as your Iterator only returns the first *n* entries. Some servers, for example, Netscape/iPlanet, enable you to exceed the size limit if you are authenticated as a manager.
5. Those servers that return their whole directory in one go (for example, non-paged search) typically cause memory problems on the client side. See “Handling memory problems in the LDAP Connector” on page 185.
6. When **Connector Flags** contains the value **deleteEmptyStrings**, then for each attribute, the LDAP Connector removes empty string values. This possibly leaves the attribute with no values (for example, empty value set). If an attribute has an empty value set then a modify operation deletes the attribute from the entry in the directory. An add operation never includes an empty attribute since this is not permitted. Otherwise, modify entry replaces the attribute values.

7. When performing a **rootdse** search in Lookup mode using the "baselevel"search scope, you must add a Link Criteria specifying that the value of **objectClass** is * (objectClass equals *) and leave the Search Base field blank. In Iterator mode the same thing is achieved by leaving the Search Base blank and setting the Search Filter to "objectClass=*".
8. When performing a normal search in Lookup mode using the "baselevel"search scope, you need to add a valid Link Criteria in accordance with the specified Search Base (for example, Search Base: cn=MyName,o=MyOrganization,c=MyCountry ; Link Criteria: sn equals MySurName).

Detect and handle modrdn operation

Some changelog connectors (the IDS Changelog Connector, Sun Directory Change Detection Connector and zOS Changelog Connector) can detect *modrdn* operations as the underlying LDAP servers' changelogs provide it. When this happens the Changelog Connector tags the Entry with the *modify* operation. The changelog attributes contain the "newrdn" attribute when the operation is modrdn. The LDAP Connector detects in its modEntry method if the "newrdn" attribute exists and if so, it replaces the rdn in the target $dn with the new value and does a context rename operation.

Note: LDAP configurations in Delta mode before Tivoli Directory Integrator v7.0 have treated modrdn operation as generic and have not handled it at all. Now they will handle it as *modify*. Also, such configurations will rename $dn if the "newrdn" attribute is provided.

Configuration

The Connector needs the following parameters; not all parameters are available or visible in all modes:

LDAP URL
The LDAP URL for the connection (ldap://*host*:*port*).

Login username
The distinguished name used for authentication to the server.

Login password
The credentials (password).

Search Base
The search base to be used. Specify a distinguished name. Some directories enable you to specify a blank string which defaults to whatever the server is configured to do. Other directory services require this to be a valid distinguished name in the directory. The default value is "<o=orgname>".

Search Filter
The search filter to be used when iterating the directory. This parameter is only used in Iterator mode, but is visible in all modes to help with schema discovery.

The button marked "**...**" to the right of the **Search Filter** field presents a Link Criteria dialog where you can fill out a link criteria form and generate the LDAP search filter.

Use the **Add** button to add more rows to build your selection criteria. The **Match Any** checkbox will generate an OR expression rather than the default AND expression. Note that this is a one-way helper. Anything you already have in the configuration will be replaced by the generated expression.

Search Scope
This parameter is not used if the Connector is in AddOnly mode. The possible values are:

subtree
Return entries on all levels from search base and below.

onelevel
Only return entries that are immediately below search base.

baselevel
Only return the entry specified by the search base.

The default value is subtree.

Size Limit
A search or iteration must return no more than this number of Entries. **0 = no limit**.

Time Limit
Searching for Entries must take no more than this number of seconds. **0 = no limit**.

Page Size
If specified, the LDAP Connector tries to use paged mode search. Paged mode causes the directory server to return a specific number of entries (called pages) instead of all entries in one chunk. Not all directory servers support this option. The default value is 0, which indicates that paged mode is disabled.

Sort Attribute
A parameter to specify server side sorting. Does not work with Netscape/iPlanet 4.2.

Note: Increases the strain on the server.

Authentication Method
Type of LDAP authentication. Can be one of the following:

- **Anonymous** - If this authentication method is set then the server, to which a client is connected, does not know or care who the client is. The server allows such clients to access data configured for non-authenticated users. The Connector automatically specifies this authentication method if no username is supplied. However, if this type of authentication is chosen and **Login username** and **Login password** are supplied, then the Connector automatically sets the authentication method to Simple.
- **Simple** - using **Login username** and **Login password**. Treated as anonymous if **Login username** and **Login password** are not provided. Note that the Connector sends the fully qualified distinguished name and the client password in cleartext, unless you configure the Connector to communicate with the LDAP Server using the SSL protocol.
- **CRAM-MD5** - This is one of the SASL authentication mechanisms. On connection, the LDAP Server sends some data to the LDAP client (that is, this Connector). Then the client sends an encrypted response, with password, using MD5 encryption. After that, the LDAP Server checks the password of the client. CRAM-MD5 is supported only by LDAP v3 servers. It is not supported against any supported versions of Tivoli Directory Server.
- **SASL** - The client (this Connector) will use a Simple Authentication and Security Layer (SASL) authentication method when connecting to the LDAP Server. Operational parameters for this type of authentication will need to be specified using the **Extra Provider Parameters** option; for example, in order to setup a DIGEST-MD5 authentication you will need to add the following parameter in the Extra Provider Parameters field:

  ```
  java.naming.security.authentication:DIGEST-MD5
  ```

  For more information on SASL authentication and parameters see: http://java.sun.com/products/jndi/tutorial/ldap/security/sasl.html.

  Note: Not all directory servers support all SASL mechanisms and in some cases do not have them enabled by default. Check the documentation and configuration options for the directory server you are connecting to for this information.

Use SSL
If this is checked, use Secure Sockets Layer for communication with the LDAP server.

Referrals

Specifies how referrals encountered by the LDAP server are to be processed. The possible values are:

- **follow** – Follow referrals automatically
- **ignore** – Ignore referrals
- **throw** – Throw a ReferralException when a referral is encountered. You need to handle this in an error Hook.

Connector Flags

Flags to enable specific behavior.

deleteEmptyStrings

This flag causes the Connector to remove attributes containing only an empty string as value before updating the directory. If you are using an LDAP version 3 server, you must use this flag, as the value of an attribute cannot be an empty string.

Extra Provider Parameters

Additional JNDI provider parameters. The format is one colon separated *name*:*value* pair on each line.

Return attributes

List of attributes to return (one attribute per line). If you leave this empty, all non-operational (user) attributes are returned. Any operational attributes (such as modifyTimestamp) must still be listed explicitly in order to be returned.

Binary Attributes

A list of attributes that are treated as binary. The format is one attribute name on each line. If this is not specified, a default list of attributes is used. The default list is:

- photo
- personalSignature
- audio
- jpegPhoto
- javaSerializedData
- thumbnailPhoto
- thumbnailLogo
- userPassword
- userCertificate
- authorityRevocationList
- certificateRevocationList
- crossCertificatePair
- x500UniqueIdentifier
- objectGUID
- objectSid

Note: An AssemblyLine can have one list of binary attributes only. If you have several LDAP Connectors in an AssemblyLine, the last Connector must define the list of binary attributes for all the LDAP Connectors in this AssemblyLine if you need to change this from the default.

Auto Map AD Password

Used for adding or updating a user's password in Active Directory using LDAP. When checked, it maps the LDAP password (a **conn** attribute that must be called **userPassword**) to another name (**unicodePwd**). **unicodePwd** has a special format that the Connector translates into.

Note: Not needed for ADAM.

LDAP Trace File
Trace LDAP BER packets to file; this can be useful for debugging.

Sort Attribute
A parameter to specify server side sorting. Does not work with Netscape/iPlanet 4.2.

Note: This increases the strain on the server.

Virtual List View Page Size
Use Virtual List View for iterations. This might be efficient on some servers, but testing shows that some other servers (for example, Netscape/iPlanet 4.2) are very slow in this respect. However, it does provide a workaround to the out-of-memory problem. Also see "Virtual List View Control."

Simulate Rename
If the server does not support rename, simulate it with **delete** and **add** operations.

Add Attribute (instead of replace)
This option changes the default behavior of the LDAP Connector when it modifies an entry.

If this checkbox is checked, the LDAP Connector sets the constraint **DirContext.ADD_ATTRIBUTE**. If this checkbox is not checked, the LDAP Connector sets the constraint **DirContext.REPLACE_ATTRIBUTE**.

By setting the **DirContext.ADD_ATTRIBUTE** constraint for the LDAP connection, you add new values to any attribute that goes through the AssemblyLine. This might mean that the same value gets repeatedly added to the entry if not used carefully. This might also result in an exception if the attribute in question is single-valued. If **DirContext.REPLACE_ATTRIBUTE** is set, the behavior is the same as the old LDAP Connector (default behavior), that is, all values for the attribute are replaced by whatever might be in the work entry.

Comment
Your comments here.

Detailed Log
If this field is checked, additional log messages are generated.

Virtual List View Control

In order to use the Virtual List View Control in Tivoli Directory Integrator 7.1, the JNDI/LDAP Booster Pack from Sun Microsystems needs to be downloaded (http://java.sun.com/products/jndi/downloads/index.html). After downloading the Booster Pack the "ldapbp.jar" contained in the pack needs to be copied to the *TDI_install_dir*\jars folder before starting Tivoli Directory Integrator. If the Virtual List View control is used, but the "ldapbp.jar" is unavailable, the AssemblyLine will fail with a corresponding error message.

Handling memory problems in the LDAP Connector

Some servers return the whole search result in one go (for example, non paged search) and this typically causes memory problems. It might look to you that IBM Tivoli Directory Integrator leaks memory, but that is just because it is processing the entries from the server while the server continues to pour more and more entries into it.

LDAP servers such as Active Directory support the **Paged Search** extension that enables you to retrieve a page (the number of objects to return at a time), and this is the preferred way to handle big return sets (see the **Page Size** parameter for more info on this). You can always test if a server supports the paged search by clicking the button to the right of the **Page Size** parameter in the LDAP Connector Configuration tab.

If the **Page Size** parameter is not supported, you might have a problem, since there is little a client can do when being overwhelmed by the Server. Here are some workarounds:

- See the **Virtual List View Page Size** parameter that lets you do a virtual list view. This might or might not be efficient, depending on the LDAP server you use.
- If you know that your directory is a size that can be kept in memory, you can increase the memory available to the Java VM. See the appendix "Increasing the memory available to the Virtual Machine" in *IBM Tivoli Directory Integrator V7.1 Users Guide*, and take particular notice of a current issue with the LDAP Connector deployed on AIX.
- A general solution to this problem is to use a server-specific utility to dump the LDAP database to an LDIF file or some other file format and then read or iterate that file using a file or URL Connector. A command line can be started in the prolog (before Connectors activated using `system.shellCommand`), producing the LDIF export and then the AssemblyLine reads that file. It is an effective solution, when possible to implement. Remember that if you are in a mode where you iterate whole, large directories, you are able to do implement as a batch.
- In some cases you can even use IBM Tivoli Directory Integrator to dump the directory search to file. This is possible because writing quickly to a file might enable IBM Tivoli Directory Integrator to access enough of the data to keep up with the feed (depending on the amount of data and the speed of the feed). If your AssemblyLine takes too long to process an entry (for example, if it is updating another directory), the entry flood happens sooner. However, this solution is very time dependent and must be avoided if you have a better method.

Built-in rules for reconnect functionality

The Connector has implemented logic for reconnect processing as of Tivoli Directory Integrator v6.1.1 fixpack 2. The Connector-specific built-in rules makes it possible to perform a reconnect if `javax.naming.ServiceUnavailableException` is thrown regardless of the message.

In versions of Tivoli Directory Integrator before v7.1, the Connector had a loop that tried 10 times to establish the initial connection. This was to work around a problem with some servers, but it had the side effect that a failure to establish the initial connection could take a very long time if the server was down. In v7.1, this "loop" is moved into reconnect rules instead. This way you may specify if a reconnect attempt should take place, and also how many times it should be tried. For backwards compatibility, initial reconnection is enabled.

Searching against an SDBM backend on z/OS

When using the LDAP Connector for searches against an SDBM backend on z/OS, you need to consider the following:

1. When an LDAP Connector in Iterator mode is used to get a list of user profiles on an z/OS SDBM (LDAP) service, by default only the DN Attribute is returned. Other attributes are not returned even with a "*" attribute specified in the input map. This is a known limitation of the LDAP connector (it was not originally intended for this). To retrieve all the attributes, construct the AssemblyLine such that you use the LDAP Connector first in Iterator mode to retrieve the DN and subsequently use the LDAP Connector in Lookup mode with Link Criteria using the DN (that is, Link Criteria set to "$dn EQUAL $$dn").

   Note: Here a "presence" filter is used in the Iterator Connector's configuration (Config Tab-> Search Filter) to determine the scope of DN to retrieve and an subsequent equivalence filter is used in the Link Criteria in an LDAP connector in Lookup mode.
2. There are 3 user profiles for which the Iterator/Lookup flow does not work with an SDBM backend on z/OS:
   - $dn 'racfid=irrmulti,profiletype=user,sysplex=sysb'
   - $dn 'racfid=irrsitec,profiletype=user,sysplex=sysb'
   - $dn 'racfid=irrcerta,profiletype=user,sysplex=sysb'

The lookup may get the following error on these user profiles: `'ICH30001I UNABLE TO LOCATE USER'` or `'ICH31005I NO ENTRIES MEET SEARCH CRITERIA'`. This happens because these users are not real users and therefore should not be the subject of searches. The SDBM backend will do a "listuser" under the covers that issues the request in uppercase and therefore, will not find the profiles. This is expected behavior.

LDAP Connector methods (API)

This section describes some of the methods available in the LDAP Connector. The exhaustive API reference is in the JavaDocs; they can be viewed by choosing **Help -> Welcome** screen, **JavaDocs** link in the Config Editor.

LDAP compare

```
public boolean compare(String compdn, String attname, String attvalue)
  throws Exception
```

where

- *compdn* is the DN on which you want to compare an attribute.
- *attname* is the name of the attribute you want to compare.
- *attvalue* is the value for *attvalue* that you want to check comparison for.

If the value is equal, **true** is returned. If the value is not equal, the value **false** is returned. For example, if you wanted to determine if the userpassword attribute for **cn=joe,o=ibm** was equal to **secret**, use the method: `compare("cn=joe,o=ibm", "userpassword", "secret")`.

Adding a value to an attribute

This method adds a given value to an attribute:

```
public void addAttributeValue(String moddn, String modattr, String modval)
  throws Exception
```

where

- *moddn* is the DN to which you want to add the attribute value.
- *modattr* is the name of the attribute you want to add a value to.
- *modval* is the value you want to add to *modattr*.

For example, if you want to add **cn=bob** to the **members** attribute of **cn=mygroup**, use the method: `addAttributeValue("cn=mygroup", "members", "cn=bob")`

A java.langException is thrown when the underlying modify operation fails.

Replacing an attribute value

This method replaces a given value for an attribute:

```
public void replaceAttributeValue(String moddn, String modattr, String modval)
  throws Exception
```

where

- *moddn* is the DN for which you want to replace the attribute value.
- *modattr* is the name of the attribute you want to replace a value for.
- *modval* is the value you want to replace for *modattr*.

For example, if you want to replace the **members** attribute of **cn=mygroup** with only **cn=bob**, use the method: `replaceAttributeValue("cn=mygroup", "members", "cn=bob")`

A java.langException is thrown when the underlying modify operation fails.

Removing an attribute value

This method removes a given value from an attribute:

```
public void removeAttributeValue(String moddn, String modattr, String modval)
  throws Exception
```

where

- *moddn* is the DN for which you want to remove the attribute value.
- *modattr* is the name of the attribute from which you want to remove a value.
- *modval* is the value you want to remove from *modattr*.

For example, if you want to remove the value **cn=bob** from the attribute **members** in the DN **cn=mygroup**, use the method: `removeAttributeValue("cn=mygroup", "members", "cn=bob")`

A java.langException is thrown when the underlying modify operation fails.

Removing all attribute values

This method removes all values for a given attribute:

```
public void removeAllAttributeValues(String moddn, String modattr)
  throws Exception
```

where

- *moddn* is the DN from which you want to remove the attribute values.
- *modattr* is the name of the attribute from which you want to remove all values.

For example, if you want to remove all values of the **members** attribute of **cn=mygroup**, use the method: `removeAllAttributeValues("cn=mygroup", "members")`

A java.langException is thrown when the underlying modify operation fails.

Flag in Config Editor for default action for attribute add or replace

In the LDAP Connector Config Editor there is a checkbox named **Add Attributes (instead of replace)**. This option changes the default behavior of the LDAP Connector when it modifies an entry.

If this checkbox is checked, the LDAP Connector sets the constraint `DirContext.ADD_ATTRIBUTE`. If this checkbox is not checked, the LDAP Connector sets the constraint `DirContext.REPLACE_ATTRIBUTE`.

By setting `DirContext.ADD_ATTRIBUTE` constraint for the LDAP connection, you add new values to any attribute that goes through the AssemblyLine. This might mean that the same value gets repeatedly added to the entry if not used carefully. This might also result in an exception if the attribute in question is single-valued. If `DirContext.REPLACE_ATTRIBUTE` is set, the behavior is the same as the old LDAP Connector (default behavior), that is, all values for the attribute are replaced by whatever might be in the work entry.

You typically want this flag set when you are handling groups. If you want to add a **member** (a value) to a **group** (an attribute), you do not want to delete all the other values.

The old behavior was to replace the attribute with the new value. This behavior remains the default.

Note: This property can be set at any time while the Connector is running by setting the property `addAttribute` from your scripts. Use something similar to the following command:

```
work.setProperty("addAttribute", true)
```

Note: This property does not affect the behavior of the `addAttributeValue` and `replaceAttributeValue` methods described previously.

Rebind

The LDAP Connector has a `rebind()` method which facilitates building advanced solutions like virtual directories and other solutions that map incoming authentication requests (use any of the support protocols) to LDAP. See the JavaDocs for more information.

Skip Lookup in Update and Delete mode

The LDAP Connector supports the **Skip Lookup** general option in Update or Delete mode. When it is selected, no search is performed prior to actual update and delete operations. It requires a $dn parameter to be specified in order to operate properly.

See also

"JNDI Connector" on page 177,
"Active Directory Change Detection Connector" on page 8,
"Sun Directory Change Detection Connector" on page 215,
"IBM Tivoli Directory Server Changelog Connector" on page 123
"z/OS LDAP Changelog Connector" on page 291,
Wikipedia on LDAP.

LDAP Server Connector

The LDAP Server Connector accepts an LDAP connection request from an LDAP client on a well-known port set up in the configuration (usually 389). The LDAP Server Connector only operates in Server mode, and spawns a copy of itself to take care of any accepted connection until the connection is closed by the LDAP client.

This Connector can be used in conjunction with the IBM Password Synchronization plug-ins. For more information about installing and configuring the IBM Password Synchronization plug-ins, please see the *IBM Tivoli Directory Integrator V7.1 Password Synchronization Plug-ins Guide*.

Each LDAP message received on the connection drives one cycle of the LDAP Server Connector logic. The main thread returns to listening for similar LDAP requests from other LDAP clients. At this point, Attribute Mapping will take place, and the appropriate attributes like the LDAP Operation should be mapped into the *work* object.

The rest of the AssemblyLine will be executed, and when the cycle reaches the Response channel the return message is built from Attributes mapped out, and sent back to the client. If it was an LDAP search command, the user will call the **add** method to build the data structure that is to be sent back to the client. The LDAP Server Connector goes back to listening for the next LDAP command on the existing connection.

The value of the LDAP operation is provided in the **LDAP.operation** attribute in the LDAP Server Connector *conn* entry, which should be mapped into the work entry for further processing (along with any other required attributes). Legal values are **SEARCH**, **BIND**, **UNBIND**, **COMPARE**, **ADD**, **DELETE**, **MODIFY**, and **MODIFYRDN**. The LDAP message provides a number of attributes for the specified LDAP operation.

Scripting

The part of the AssemblyLine that follows the LDAP Server Connector must do work to determine the desired outcome of the LDAP message. The basic LDAP operations (**SEARCH**, **BIND**, **UNBIND**, **COMPARE**, **ADD**, **DELETE**, **MODIFY**, and **MODIFYRDN**) are provided as values in the LDAP Server Connector scripting environment to facilitate scripting, for example, if **LDAP.operation** equals **BIND**. The user code sends search result entries to the client by calling the **add (entry)** method in the LDAP Server Connector. The entry must be formatted with legal LDAP attribute names plus the special attribute **$dn** (the distinguished name of the entry).

Returning the LDAP message returned values

The user-provided code in the AssemblyLine responds to each request by setting the **ldap.status**, **ldap.matcheddn** and **ldap.errormessage** entry attributes. **ldap.matcheddn** and **ldap.errormessage** are optional.

In the Response channel phase of the AssemblyLine, the LDAP Server Connector formats and returns some of the attributes of the *work* entry. These are:

- **LDAP.status**
- **LDAP.errormessage**

Note: Only string is supported. The **resultCode** is by default set to **0** (success). A **resultCode** indicating anything other than successful must be specifically set by the user.

Error handling

The LDAP Server Connector terminates the connection and records an error if the received message does not conform to the LDAP v3 format

Note: The LDAP Server Connector does not perform any validation on the incoming attributes. Any operation or parameter value is therefore accepted.

Configuration

The Connector needs the following parameters:

LDAP Port
The TCP port on which this Connector listens. You can choose one of the default values, or provide your own port number.

Use SSL
If checked, the server connector will only accept SSL connections.

Note: Depending on your solution implementation, you may need to change the port number as well.

Character Encoding
Specify the character set here. The default is **UTF-8**.

Binary Attributes
A list of attributes that are treated as binary (a binary attribute is returned as a byte array, not a string). The format is one attribute name on each line.

Note: An AssemblyLine can have one list of binary attributes only. If you have several LDAP Connectors in an AssemblyLine, the last Connector must define the list of binary attributes for all the LDAP Connectors in this AssemblyLine (if you need to change this from the default).

Comment
A comment for your own use.

Detailed Log
If this field is checked, additional log messages are generated.

See also

"LDAP Connector" on page 181

Log Connector

The Log Connector is very different from other connectors as it does not have any idea of source/target system. This connector was written exclusively to give you an alternative access to the Tivoli Directory Integrator logging features; see "Logging and debugging" in the *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*.

Introduction

The Log Connector enables you to use logging utilities in a simpler way, requiring less scripting. The connector can be inserted at any point in the AssemblyLine and enabled/disabled dynamically. Prior to the introduction of this connector you would have to add script code that invoked the AssemblyLine's log object. Using this log object would add log messages to all loggers associated with the AssemblyLine's log object. Similarly, adding a logger to the AssemblyLine (using the AssemblyLine logging configuration screen) would also merge Tivoli Directory Integrator's internal logging to the log output channel causing the log to fill up with possibly unwanted log messages. The Log Connector gives you explicit control of all messages written to the log channel.

This Connector supports AddOnly mode only.

Schema

The schema for the Log Connector is flat with the following predefined attributes:

Table 17.

Attribute	Description
message	The message to be logged.
level	Optional level of the logged message.
exception	Optional *java.lang.Exception*

When exception is present, the level parameter is ignored and `LogInterface.error(msg, exception)` is called.

When both exception and level is absent then `LogInterface.info(msg)` is called.

Configuration

The connector configuration for the Log Connector will display the form associated with the selected logger component.

Select Logger

The "Select Logger" function in the configuration screen lets you choose a predefined configuration from the installed log components folder. This configuration is copied into the connector configuration for use with the actual logger instantiation. Available categories and choices are:

Apache Log4J loggers

This is the traditional and most feature-rich category.

Available Apache Log4J loggers are:

- ConsoleAppender
- CustomAppender
- DailyRollingFileAppender
- FileAppender
- NTEventLog

- FileRollerAppender
- SystemLogAppender
- SyslogAppender

Java Util loggers

- Category based configuration
- FileHandler (java.util.logging)

JLOG loggers

- Category based configuration
- FileHandler (com.ibm.log.FileHandler)

An alternative to define loggers is to use the built-in logging feature and a `LogConfigItem` configuration object. This method bypasses Log4J's lookup in the `log4j.properties` file; see "Creating additional Loggers" on page 604 for more information.

Logger configuration screen

Apache Log4J Loggers: The Apache Log4j logging utility is bundled with Tivoli Directory Integrator. Log4j uses a properties file to configure loggers that are used by Tivoli Directory Integrator. Obtaining a logger via the Log4J API requires a category name, which matches a logger definition in the `log4j.properties` file. By default, Tivoli Directory Integrator configures Log4J to use *`solution dir`*`/etc/log4j.properties`; see *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide* for example configurations.

Layouts:

With most loggers you can specify the layout of the output log. You have a choice of the following layouts:

- Pattern Layout – Uses a pattern to format the output message (see Patterns below)
- Simple Layout – Prints out the log level followed by " – " and the actual log message
- HTML Layout – Prints out the log in an HTML table
- XML Layout – Prints out log events according to log4j.dtd (http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/xml/doc-files/log4j.dtd)

Patterns:

Many loggers use a pattern to format the output string that goes to the log (Pattern.ConversionPattern). The format of this pattern is documented in the respective logging utility's documentation. Below is an incomplete listing of some of the more useful conversion characters that can be used in those patterns.

A pattern is a string that contains special constructs that are substituted by a computed value. Use a percentage sign (%) followed by one of the following characters to insert computed values:

Table 18.

Character	Effect
m	Used to output the log message from the caller.
c	Used to output the category of the logging event. The category conversion specifier can be optionally followed by precision specifier, this is a decimal constant in brackets.
d	Used to output the date of the logging event. The date conversion specifier may be followed by a date format specifier enclosed between braces. For example, %d{HH:mm:ss,SSS} or %d{dd MMM yyyy HH:mm:ss,SSS}. If no date format specifier is given then ISO8601 format is assumed.

Table 18. (continued)

Character	Effect
F	Used to output the file name where the logging request was issued.
n	The platform specific end-of-line character(s)
p	The priority of the logged event
t	The name of the Thread that generated the log event (for example, AssemblyLine name etc)
%	Outputs a single percent sign

Category based configuration: The category based configuration logger will delegate creating the logger to Log4J. Log4j will use the `log4j.properties` file to find a match for the category and return a logger as defined in the properties file.

Category
Enter the category to use. The default is empty.

ConsoleAppender: The console appender writes to the standard output/error streams. Parameters are:

Layout
Select a Layout for the Appender. Available values are:
- Pattern (the default)
- Simple
- HTML
- XML

Pattern
Specifies the substitution mask that defines log formatting. Only used when Layout is Pattern. You can select from the following pre-defined patterns, or specify your own:
- %d{ISO8601} %-5p [%c] - %m%n (the default)
- %d{HH:mm:ss} %p [%t] - %m%n
- %p [%t] %c %d{HH:mm:ss,SSS} - %m%n

CustomAppender: The custom appender is available when custom appenders have been defined in one or more java properties. Properties that start with `custom.appender.` are expected to have a value that specify an appender class that implements `com.ibm.di.log.CustomAppenderInterface`. Parameters are:

Appender Parameters
The parameter's format is not defined and it is up to Appender implementation to parse this text field. By default, this field is empty.

DailyRollingFileAppender: The daily rolling file appender rotates the log file every day. When the output file is rolled it is given a name consisting of the base name plus a date pattern string (for example, `filename.yyyy-mm-dd`.) Parameters are:

File Path
Specify the path name for the base log file. The default value is empty.

Append to file
Check this box to append to the file when the log is initialized, unchecked to overwrite. The default value is unchecked.

Date Pattern
Specifies the Date pattern that logfile names are suffixed with, and which controls when rollover occurs (i.e. when the filename changes.) You can select from the following pre-defined values or specify your own:

- .'yyyy-MM
- '.'yyyy-MM-dd
- '.'yyyy-MM-dd-HH
- '.'yyyy-MM-dd-HH-mm
- '.'dd-MM-yyyy

The default value is: '.'yyyy-MM-dd

Layout
Select a Layout for the Appender. Available values are:
- Pattern (the default)
- Simple
- HTML
- XML

Pattern
Specifies the substitution mask that defines log formatting. Only used when Layout is Pattern. You can select from the following pre-defined patterns, or specify your own:
- %d{ISO8601} %-5p [%c] - %m%n (the default)
- %d{HH:mm:ss} %p [%t] - %m%n
- %p [%t] %c %d{HH:mm:ss,SSS} - %m%n

Character Encoding
Specifies the output character encoding (for example, UTF-8); the default value is empty.

FileAppender: The file appender writes messages to an output file. Parameters are:

File Path
Specifies the file path for the log file. The default value is empty.

Append to file
Check this box to append to the file when the log is initialized, unchecked to overwrite. The default value is unchecked.

Layout
Select a Layout for the Appender. Available values are:
- Pattern (the default)
- Simple
- HTML
- XML

Pattern
Specifies the substitution mask that defines log formatting. Only used when Layout is Pattern. You can select from the following pre-defined patterns, or specify your own:
- %d{ISO8601} %-5p [%c] - %m%n (the default)
- %d{HH:mm:ss} %p [%t] - %m%n
- %p [%t] %c %d{HH:mm:ss,SSS} - %m%n

Character Encoding
Specifies the output character encoding (for example, UTF-8); the default value is empty.

NTEventLog: The NT event logger writes messages to the Windows NT event log. Parameters are:

Source
Usually the title of the application doing the logging. The default value is "itdi".

Layout

Select a Layout for the Appender. Available values are:

- Pattern (the default)
- Simple
- HTML
- XML

Pattern

Specifies the substitution mask that defines log formatting. Only used when Layout is Pattern. You can select from the following pre-defined patterns, or specify your own:

- %d{ISO8601} %-5p [%c] - %m%n (the default)
- %d{HH:mm:ss} %p [%t] - %m%n
- %p [%t] %c %d{HH:mm:ss,SSS} - %m%n

FileRollerAppender: The file roller will rotate its logs every day using a sequence number of 1 through **Number of backup files**. Parameters are:

File Path

This logger will write to <File Path>.1 and increment the extension of existing logfiles up to the specified number of backup files. The default value is empty.

Number of backup files

Specify the number of files to keep before removing the oldest log file. The default value is 5.

Layout

Select a Layout for the Appender. Available values are:

- Pattern (the default)
- Simple
- HTML
- XML

Pattern

Specifies the substitution mask that defines log formatting. Only used when Layout is Pattern. You can select from the following pre-defined patterns, or specify your own:

- %d{ISO8601} %-5p [%c] - %m%n (the default)
- %d{HH:mm:ss} %p [%t] - %m%n
- %p [%t] %c %d{HH:mm:ss,SSS} - %m%n

Character Encoding

Specifies the output character encoding (for example, UTF-8); the default value is empty.

SystemLogAppender: The system log appender writes to log files found under system_logs/{ConfigId}/{AL,EH}_X (where X is the name of the AL/EH being run). Parameters are:

Pattern

Specifies the substitution mask that defines log formatting. Only used when Layout is Pattern. You can select from the following pre-defined patterns, or specify your own:

- %d{ISO8601} %-5p [%c] - %m%n (the default)
- %d{HH:mm:ss} %p [%t] - %m%n
- %p [%t] %c %d{HH:mm:ss,SSS} - %m%n

Character Encoding

Specifies the output character encoding (for example, UTF-8); the default value is empty.

SyslogAppender: The syslog appender writes to a syslogd deamon. The syslogd deamon is the standard logging utility on most UNIX systems. Parameters are:

Host name IP Address
Specifies the hostname or IP address of the syslog daemon. The default value is 127.0.0.1

Syslog Facility
Specify the facility name to use for logged messages. Available values are:

- kern
- user
- mail
- daemon
- auth
- syslog
- lpr
- news
- cron
- authpriv
- ftp
- local0
- local1
- local2
- local3
- local4
- local5
- local6
- local7

The default value is **local7**.

Print Facility String
Check if the facility should be printed. By default, this is checked.

Layout
Select a Layout for the Appender. Available values are:

- Pattern (the default)
- Simple
- HTML
- XML

Pattern
Specifies the substitution mask that defines log formatting. Only used when Layout is Pattern. You can select from the following pre-defined patterns, or specify your own:

- %d{ISO8601} %-5p [%c] - %m%n (the default)
- %d{HH:mm:ss} %p [%t] - %m%n
- %p [%t] %c %d{HH:mm:ss,SSS} - %m%n

Java Util Loggers: These loggers are part of the standard Java VM. The layouts in java util loggers are termed formatters. Tivoli Directory Integrator includes support for the following formatters:

- Simple – Prints a brief summary of the log event in a human readable format. The summary will typically be 1 or 2 lines.
- XML – Prints the log file in the format specified by the Java Logging API (see appendix A for a complete description of the DTD).

Category based configuration: The category based configuration logger will delegate creating the logger to Java Util Logging. JUL will use its `lib/logging.properties` file to find a match for the category and return a logger as defined in the properties file.

Category
Enter the category to use. The default is empty.

FileHandler: The file appender writes messages to an output file. If the "limit" parameter sets an upper limit of the file's size, the log file is rotated when it reaches the maximum size and a new file is created to continue logging. Parameters are:

File Name
Specify the pattern for the log file name (see java.util.logging.FileHandler.)

Append
Check this box to let the logger append to the file when the log is initialized. The default is unchecked, which will cause an existing file to be overwritten.

Formatter
Select the formatter to use. Available values are:

- Simple
- XML

Limit Specifies the maximum size of the file before logging is rotated. The default is empty, which signifies infinite size.

Count Specifies the number of files to keep after a rotation of the log has occurred. The default is empty, which actually means only 1 file is used, with no rotations.

JLOG Loggers: JLOG loggers are bundled with Tivoli Directory Integrator. The layouts in JLOG are termed formatters. Tivoli Directory Integrator includes support for the following formatters:

- Simple
- CBE101Formatter – Formats log events as Common Base Event v1.0.1 XML entries.
- EnhancedFormatter – Formats log events by printing each field in the log event on a separate line, for example:

```
Date:    1999.07.16  11:20:56.842
Class:   com.ibm.log.samples.LogSample
Method:  messageSample
```

Category based configuration: The category based configuration logger will delegate creating the logger to JLOG. JLOG will use its `jlog.properties` file to find a match for the category and return a logger as defined in the properties file.

Category
Enter the category to use. The default is empty.

FileHandler: The file appender writes messages to an output file. Parameters are:

File Path
Specify the file path for the log file. The default is empty.

Append to file
Check this box to let the logger append to the file when the log is initialized. The default is unchecked, which will cause an existing file to be overwritten.

Formatter
Select the formatter to use for this logger. Available values are:

- CBE101
- Enhanced

- Simple

Lotus Notes Connector

See “Lotus Notes Connector” on page 77, in the combined Lotus Notes section.

Mailbox Connector

This Mailbox Connector provides access to internet mailboxes (POP3 or IMAP). The Mailbox Connector can be used in AddOnly, Iterator, Lookup, Update and Delete modes. The Mailbox Connector uses predefined attribute names for the headers that are used most often. If you need more than this,f use the **mail.message** property to retrieve the native message object.

On initialization, the Connector gets all available mail messages from the mailbox on the server and stores them into an internal Connector buffer. Later the Connector retrieves the messages one by one on each `getNextEntry()` call; that is, on each Iteration. When all the messages from the buffer have been retrieved, the parameter **Poll Interval** governs what happens next; see "Configuration." This is different from earlier implementations of this Connector.

If the IMAP protocol is specified the Mailbox Connector registers for notifications for messages added and messages removed from the mailbox on the server. When a notification that a message has been added to the mailbox is received, the Connector adds this message to its internal buffer. If a notification that a message has been removed from the mailbox is received, the Connector removes this message from its internal buffer.

For all supported modes except Addonly (Iterator, Update, Lookup, Delete) the Mailbox connector iterates on all folders in the mailbox, if no folder is specified in the configuration. Otherwise, there is the option to iterate on subfolders of the supplied folder. In both cases you can specify a list of comma-separated folders to be excluded when browsing the mailbox.

Notes:

1. Only one connection per user ID is supported. If the user fails to disconnect when using the schema tab, and then runs the AssemblyLine, this results in a connection refused error.
2. The Mailbox Connector does not support the Advanced Link Criteria (see "Advanced link criteria" in *IBM Tivoli Directory Integrator V7.1 Users Guide*).

Configuration

Mail Server
: The POP/IMAP mail server hosting the mailbox. It might include a port number separated by a space (*url port*). For example:

```
domino.raleigh.ibm.com 110
```

Use SSL
: When checked, the Connector uses SSL connections. When unchecked, the Connector uses non-SSL connections.

Mail Protocol
: Specify **pop3** or **imap**.

Username
: The user name.

Password
: The password for **Username**.

Mail Folder
: Specifies the name of the user's mail folder on the mail server.

  The user's mail folder stores the user's mail messages on the mail server. When using the POP3 mail protocol you must specify "INBOX" as the value for this parameter. When using the IMAP mail protocol you can specify any mail folder which exists on the mail server.

Create Folder
Check this box to create a mailbox, specified in the parameter **Mail Folder**, in case it does not exist. Applicable only in AddOnly Mode.

Get subfolders
Specifies whether the Connector in Iterator mode will iterate through the subfolders of the specified Folder. Relevant for all modes except Addonly Mode.

Exclude folders
Specifies a comma-separated list of folders that will be excluded when iterating on the mailbox. Relevant for all modes except Addonly Mode.

Poll Interval (seconds)
Specifies the amount of seconds that the Connector will sleep before polling the mail server for new mail messages.

After the AssemblyLine consumes all mail messages stored in the Mailbox Connector buffer, the Connector sleeps for a while and then reconnects to the mail server and checks for new messages. In other words, the Connector polls the server for new mail messages.

A special value of "-1" means that the Connector will not poll for new mail messages after the initial poll. This means that the AssemblyLine will terminate after it has consumed all messages retrieved by the Connector on the initial poll.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Schema

Input Map

The Mailbox Connector uses the following predefined attributes and properties, which are available in the Input Map:

mail.from
The **From** header

mail.to
The **To** (recipient) headers

mail.cc
The CC recipient headers

mail.replyto
The mail address to reply to

mail.subject
The subject header

mail.messageid
The message ID header

mail.messagenumber
The message's internal number

mail.sent
The date the message was sent

mail.received
The date the message was received

mail.body
In case of a single part message this attribute contains the message body

mail.bodyparts
In case of a multipart message this attribute contains a javax.mail.Part object.

mail.message
This is the javax.mail.Message representing the message returned in the entry.

mailbox.message
This is the `javax.mail.Message` representing the message returned in the entry. This is the same object as the one stored in **mail.message**. This Attribute ensures backward compatibility with the obsolete Mailbox EventHandler.

mail.originator
The Connector object.

event.originator
The Connector object. This is the same object as the one stored in **mail.originator**. This Attribute ensures backward compatibility with the obsolete Mailbox EventHandler.

mail.session
The Java session object (javax.mail.Session).

mailbox.session
The Java session object (javax.mail.Session). This is the same object as the one stored in **mail.session**. This Attribute ensures backward compatibility with the obsolete Mailbox EventHandler.

mail.store
The message store object (javax.mail.Store).

mailbox.folder
The folder object (javax.mail.Folder). This is the same object as the one stored in **mail.folder**. This Attribute ensures backward compatibility with the obsolete Mailbox EventHandler.

mail.operation
The operation related to mail.message. For POP3 connections only *existing* entries are reported. For IMAP connections this property contains the value *new* or *deleted*.

mailbox.operation
The operation related to mailbox.message. This is the same object as the one stored in **mail.operation**. This Attribute ensures backward compatibility with the obsolete Mailbox EventHandler.

Output Map

The Mailbox Connector gets the following Attributes from the work Entry (Output Attribute Map):

mail.from
The **From** header

mail.to
The **To** (recipient) headers

mail.subject
The subject header

mail.messageid
The message ID header

mail.messagenumber
The message's internal number

mail.addMessage
Holder for `javax.mail.Message` or `javax.mail.Message[]` object, which is used in AddOnly mode to add messages to the specified mailbox folder.

Flag.Answered
The javax.mail.Flags.Flag.ANSWERED Boolean value

Flag.Deleted
The javax.mail.Flags.Flag.DELETED Boolean value

Flag.Draft
The javax.mail.Flags.Flag.DRAFT Boolean value

Flag.Recent
javax.mail.Flags.Flag.RECENT Boolean value

Flag.Seen
The javax.mail.Flags.Flag.SEEN Boolean value

Using the Connector

Iterator Mode

In this mode the Connector is iterating through all messages from the specified folder (INBOX by default). Each message will be translated into an entry and its attributes will be available for the next step of the flow.

Depending on the backend mail server you are connecting to, you might not be able to interact directly with the body of a message. This is because the server supports multi-body parts, as opposed to a single one. In this case all the body parts can be accessed as a multi-valued attribute (for example,

```
work.getAttribute("mail.bodyparts").getValue(N).getContent();
```

where *N* is the number of the message body, the number zero indicating the first message body). In case the server does not provide multi-body parts then the message body will be in the attribute *mail.body*.

If no folder is specified, then the Connector iterates through all mailbox folders. A parameter allows for configuring that certain folders can be skipped when iterating. The names of these folders are entered in a text box separated by comma. Another parameter defined as a checkbox is used to indicate whether the Connector also should iterate through the subfolders of the specified folder. Note that when POP3 is chosen this option is not available, since the POP3 provider supplies a single folder – "INBOX". If the **Mail Folder** parameter is left empty, the Connector will iterate on the messages in the INBOX folder.

Lookup Mode

In this mode the retrieved Entry(s) are based on the LinkCriteria defined for the connector. In case no message is found then an exception is thrown. This mode is similar to Iterator mode but here you can not iterate through the messages in the mail folders unless you have LinkCriteria defined. The attributes mapped in the work entry will be filled in if a single message is found. If more than one entry is returned then the AssemblyLine execution will stop; you can work around this by providing logic for cases like this in the 'On Multiple Entries' Hook.

When the Mailbox Connector is used in Lookup mode the only searchable headers are:

- mail.from
- mail.to
- mail.cc
- mail.subject
- mail.messageid
- mail.messagenumber

Delete Mode

In this mode the entry(s) returned are based on the defined LinkCriteria. In one message is found then it is deleted. If no messages are found then either 'On No Match' Hook is called (if defined) or an exception is thrown and execution stops. If more than one message is found then either 'On Multiple Entries' Hook is called (if defined) or exception is thrown.

When the Mailbox Connector is used in Delete mode the only searchable headers are:

- mail.from
- mail.to
- mail.cc
- mail.subject
- mail.messageid
- mail.messagenumber

AddOnly Mode

AddOnly mode is used for putting messages into a specified mailbox folder. For this purpose you must first configure the mail server, mail credentials, protocol type (only IMAP) and the folder in which the new messages will be delivered.

This mode can only be used with IMAP protocol since the POP3 protocol does not support appending of messages. For more information about the restrictions of the POP3 provider for the Java Mail API refer to: http://java.sun.com/products/javamail/javadocs/com/sun/mail/pop3/package-summary.html.

In case the folder defined in the configuration of the Connector does not exist, the parameter "createFolder" is taken into account. If it is checked and the supplied folder is not present, a new one with this name is created. The new Messages are delivered to the connector in an attribute called *mail.addMessage*; this attribute is passed an Object of type `javax.mail.Message` or an array of that type. The Connector connects to the mail box and appends the message(s) into the specified folder.

Update Mode

In update mode the Mailbox Connector is able to make changes to the flags of a message in the specified mail folder. The supported flags are: Answered, Deleted, Draft, Recent and Seen. These parameters are passed to the Connector as Attributes in its output map. Flags and therefore the corresponding Attributes are of Boolean type. The Flags can be manipulated through the `javax.mail.Message.setFlag(...)` method.

You specify in the work entry which flag should be updated and the new value. In case the message store does not support the flag that you want to update, a message is logged containing the flag attribute for which the operation failed. Afterwards the connector continues with the updates of the other flags.

When the Mailbox Connector is used in Update mode the only searchable headers are:

- mail.from
- mail.to
- mail.cc
- mail.subject
- mail.messageid
- mail.messagenumber

Memory Queue Connector

The Memory Queue (MemQueue) Connector provides a connector-like functionality to read and write to the memory queue feature (aka. MemBufferQ). This is an alternative to writing script to access a memory queue and is an extension of the "Memory Queue Function Component" on page 402 (function component).

The objects used to communicate between components are not persistent and are not capable of handling large return sets. For example, large data returned by an *ldapsearch* operation. In order to solve this problem, an internal threadsafe memory queue can be used as a communications data structure between AL components. It can contain embedded logic that would trigger whenever buffer is x% full/empty/data available.

There can be multiple readers and writers for the same queue. Every writer has to obtain a lock before adding data. The writer has to release lock before a reader can access it. Connectors in Iterator mode have a parameter that determines when the read lock is released – **After single read**, on **AL cycle end** or **Connector close**.

This Connector supports AddOnly and Iterator modes only.

Notes:

1. Because of the non-persistent nature of this Connector, we recommend that you use the "System Queue Connector" on page 227 instead, because that Connector relies on the underlying Java Messaging Service (JMS) functionality with persistent object storage.
2. When the Memory Queue Connector is in Iterator mode it reads from the configured queue. If that queue does not exist it is created. If you don't want this behavior, you need to set the system property `tdi.memq.create.queue.default=false`, in this case Tivoli Directory Integrator will behave like previous versions; this implies that when the queue does not exists, an exception is thrown in Iterator Mode.

This Connector can also be used in connection with MemQueue pipes set up from JavaScript, although it is important to note that a MemQueue pipe created by the MemQueue Connector will be terminated when the Connector closes.

The Memory queue buffer is a FIFO type of data structure, where adding and reading can occur simultaneously. It works as a pipe where additions happen at one end and reading happens at the other end and reading removes the data from queue.

The Memory queue buffer provides overflow storage using the System Store when a threshold value is reached, which is a function of the runtime memory available.

Memory queue components

Paged memory buffer queue
A queue type buffer having the following functions:

1. Read and write
2. Finding the size
3. Registering and unregistering callback triggers
4. Generating triggers by calling callback methods
5. Option for setting the system store to use for paging

Watermark
This is the maximum size of the queue; refer to the configuration page for operational details.

Pages The connector is able to buffer a set of objects before writing them to the System Store. The buffer it uses is called a page. Users can specify the number of the objects each page may contain. This

is done using the pagesize parameter. Using pages helps the connector more efficiently read/write objects from/to the System Store. This option is used only used when paging is used; when paging support is switched off, the queue is not divided into pages, but is purely a sequence of elements.

Threshold
This is the number of pages that can exist in the memory and beyond which pages must be flushed to the system store. This is calculated depending on the page size entered by the user and runtime available memory.

System Store
Database that stores the pages when the threshold is reached.

Global Lookup Table
A global table that can store memory buffer objects. This is to support a named pipe mechanism of sharing between threads. A thread can lookup a memory buffer queue using a name in the table, if it exists a reference to the queue object is returned to the user else a new queue object is created with that name and added to the GLT.

High level workflow

The Memory queue Buffer is a queue of pages containing objects. When a particular threshold (the "watermark") is reached, a new thread is created that starts writing to another buffer of pages; when a page is full, it transfers the page to either to the main queue or to the system store. When a page is read from the main queue, one page is transferred from system store to the main queue; in doing it also deletes that page from the system store.

Configuration

Instance
Name of the Config instance. Current instance is assumed if it is null.

Queue
Name of the queue or pipe which is to be created.

Read timeout
The interval in milliseconds to wait for, before control returns, if no entries were found in the queue.

Iterator Read Lock Release
When the connector is in iterator mode, this determines when the read lock on the specified memory queue will be released. You can select one of these values: **On Single Read** (default), **AL cycle end** and **Connector close**.

Percent memory to use
This determines what percentage of memory can be utilized by the memory queue. The default value is 50.

Watermark
This is the threshold at which objects are persisted to the System Store. Note that the **Page Size** determines when pages are actually written, so the Watermark should be a multiple of the Page Size.

Page Size
Number of entries in one page. The default value is 100.

Database name
System Store database name: a JDBC URL to a System Store database (or blank for the default System Store).

Username
Username to connect to the System Store database.

Password
Password to use when signing on to the database.

Table name
Name of System Store table to use for paging.

Detailed Log
Check for detailed log messages.

Accessing the Memory Queue programmatically

The Memory Queue can be accessed directly from JavaScript, not only through the Connector.

1. To create new pipe - There are two methods for this.
   a. Paging disabled - **newPipe**(String instName,String pipeName,int watermark) // Does not require any DB related entries
   b. Paging enabled - newPipe(String instName,String pipeName,int watermark,int pagesize) // Requires DB initialization

   An example script with paging enabled:

   ```
   var memQ=system.newPipe( "inst","Q1",1000,10) ;
   memq.initDB(dbName, jdbcLogin, jdbcPassword, tableName);  // Required to Initialize DB
   memQ.write(conn);
   ```

2. **getPipe**(String instName,String pipeName)
3. **purgeQueue()**

   An example script would look something like this:

   ```
   var q =system.getPipe("Inst1","Q1") ;
   q.purgeQueue();
   ```

4. **deletePipe**(String pipeName)

   Example:

   ```
   var q =system.getPipe("Inst1","Q1");
   q.deletePipe();
   ```

The following is an example script to read from the Memory Queue using API calls:

```
var memQ=system.getPipe( "inst","Q1") ;
var size=memQ.size();

for(var count=0;i<=size;count++){
 main.logmsg(memQ.read());
}
```

Memory Stream Connector

The Memory Stream Connector can read from or write to any Java stream, but is most often used to write into memory, where the formatted data can be retrieved later. The allocated buffer is retrieved/accessed as needed.

Note: The memory stream is confined to the local JVM, so it's not possible to interchange data with a task running in another JVM; be it on the same machine or a different one.

The Connector can only operate in Iterator mode, AddOnly mode, or Passive state. The behavior of the Connector depends on the way it has been initialized.

initialize(null)
This is the default behavior. The Connector writes into memory, and the formatted data can be retrieved with the method getDataBuffer(), only available in Memory Stream Connectors. Assuming the Connector is named MM, this code can be used anywhere (for example, Prolog, Epilog, all Hooks, script components, and even inside attribute mapping):

```
var str = MM.connector.getDataBuffer();
// use str for something.
// To clear the data buffer and ready the Connector
  for more output, re-initialize
MM.connector.initialize(null);
```

initialize(Reader r)
The Connector reads from **r**. This can be used if you want to read from a stream.

initialize(Writer w)
The Connector writes to **w**.

initialize(Socket s)
The Connector can both read from and write to a Socket **s**.

Note: Do not reinitialize unless you want to start reading from or writing to another data stream. If you want to use the Connector Interface object, see "The Connector Interface object" on page 524. This Connector has an additional method, the getDataBuffer() method.

Configuration

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Parser The name of a Parser to format the output or parse the input.

Sun Directory Change Detection Connector

The Sun Directory Change Detection Connector is a specialized instance of the LDAP Connector; this connector was previously called the Netscape/iPlanet Changelog Connector.

In Sun/iPlanet Directory Server 5.0, the format of the changelog was modified to a proprietary format. In earlier versions of iPlanet Directory Server, the change log was accessible through LDAP. Now the changelog is intended for internal use by the server only. If you have applications that must read the changelog, you will need to use the iPlanet Retro Change Log Plug-in for backward compatibility.

Since it is not always possible to run the Sun/iPlanet Directory Server in Retro Changelog mode, the Connector is able to run in two different Delivery Modes:

1. *Changelog* mode – in this mode the Connector will iterate trough the changelog (enabled by the iPlanet Retro Change Log Plug-in) and after delivering all Entries it will poll for new changes or use change notifications
2. *Realtime* mode – in this mode, only changes received as notifications will be delivered and offline changes will be lost. The Connector will not use the changelog in this mode. This delivery mode is necessary for Sun/Netscape/iPlanet Servers that do not support a changelog

This Connector supports Delta Tagging, in two different operation modes:

- In **Changelog** mode Delta tagging is supported at the Entry level, the Attribute level and the Attribute Value level. It is the LDIF Parser that provides delta support at the Attribute and Attribute Value levels.
- In **Realtime** mode Delta tagging will be performed at the Entry level only.

The Connector will detect *modrdn* operations in the Server's changelog, see “Detect and handle modrdn operation” on page 182 for more information.

Note: This component is not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.

Attribute merge behavior

In older versions of Tivoli Directory Integrator, in the Sun Directory Change Detection Connector merging occurs between Attributes of the changelog Entry and changed Attributes of the actual Directory Entry. This creates issues because you cannot detect the attributes that have changed. The Tivoli Directory Integrator 7.1 version of the Connector has logic to address these situations, configured by a parameter: **Merge Mode**. The modes are:

- **Merge changelog and changed data** - The Connector merges the attributes of the Changelog Entry with changed attributes of the actual Directory Entry. This is the older implementation and keeps backward compatibility.
- **Return only changed data** - Returns only the modified/added attributes and makes Changelog Iterator and Delta mode easier. This is the default; note that in configurations developed under and migrated from earlier versions of Tivoli Directory Integrator, you may need to select **Merge changelog and changed data** manually so as to ensure identical behavior.
- **Return both** - Returns an Entry which contains changed attributes of the actual Directory Entry and an additional attribute called "changelog" which contains attributes of the Changelog Entry. Allows you to easily distinguish between two sets of Attributes.

Delta tagging is supported in all merge modes and entries can be transferred between different LDAP servers without much scripting.

Note that in **Realtime** mode when the LDAP search base is different than "cn=changelog", the Connector cannot determine which attributes of Directory Entry are changed so no matter what value the Merge

Mode parameter has, the output entry will still be the same. Of course, in Realtime mode when the server supports changelog and search base is set to "cn=changelog" the output entry is merged according to the chosen Merge Mode.

Configuration

The Connector needs the following parameters:

LDAP URL
The LDAP URL for the connection (ldap://*host*:*port*).

Login username
The LDAP distinguished name used for authentication to the server. Leave blank for anonymous access.

Login password
The credentials (password).

Iterator State Key
Specifies the name of the parameter that stores the current synchronization state in the User Property Store of the IBM Tivoli Directory Integrator. This must be a unique name for all parameters stored in one instance of the IBM Tivoli Directory Integrator User Property Store.

Pressing the **Delete** button causes this state information to be deleted from the User Property Store.

Start at changenumber
Specifies the starting changenumber. Each Changelog entry is named **changenumber=intvalue** and the Connector starts at the number specified by this parameter and automatically increases by one. The special value **EOD** means start at the end of the Changelog.

Note that this parameter is only used when the Iterator State is blank or not saved.

Pressing the **Query** button causes the first and last change numbers to be retrieved from the Server.

Authentication Method
Type of LDAP authentication. Can be one of the following:

- **Anonymous** - If this authentication method is set then the server, to which a client is connected, does not know or care who the client is. The server allows such clients to access data configured for non-authenticated users. The Connector automatically specifies this authentication method if no username is supplied. However, if this type of authentication is chosen and **Login username** and **Login password** are supplied, then the Connector automatically sets the authentication method to Simple.
- **Simple** - using **Login username** and **Login password**. Treated as anonymous if **Login username** and **Login password** are not provided. Note that the Connector sends the fully qualified distinguished name and the client password in cleartext, unless you configure the Connector to communicate with the LDAP Server using the SSL protocol.
- **CRAM-MD5** - This is one of the SASL authentication mechanisms. On connection, the LDAP Server sends some data to the LDAP client (that is, this Connector). Then the client sends an encrypted response, with password, using MD5 encryption. After that, the LDAP Server checks the password of the client. CRAM-MD5 is supported only by LDAP v3 servers. It is not supported against any supported versions of Tivoli Directory Server.
- **SASL** - The client (this Connector) will use a Simple Authentication and Security Layer (SASL) authentication method when connecting to the LDAP Server. Operational parameters for this type of authentication will need to be specified using the **Extra Provider Parameters** option; for example, in order to setup a DIGEST-MD5 authentication you will need to add the following parameter in the Extra Provider Parameters field:

  ```
  java.naming.security.authentication:DIGEST-MD5
  ```

For more information on SASL authentication and parameters see: http://java.sun.com/products/jndi/tutorial/ldap/security/sasl.html.

Note: Not all directory servers support all SASL mechanisms and in some cases do not have them enabled by default. Check the documentation and configuration options for the directory server you are connecting to for this information.

Use SSL
If Use SSL is **true**, the Connector uses SSL to connect to the LDAP server. Note that the port number might need to be changed accordingly.

ChangeLog/Notifications Base
Specifies the search base where the Changelog is kept. The standard DN for this is **cn=changelog**. Also known as Notification Context for 'Realtime' Delivery Mode.

Extra Provider Parameters
Allows you to pass a number of extra parameters to the JNDI layer. It is specified as name:value pairs, one pair per line.

State Key Persistence
Governs the method used for saving the Connector's state to the System Store. The default and recommended setting is **End of Cycle**, and choices are:

After read
Updates the System Store when you read an entry from the Sun Directory Server change log, before you continue with the rest of the AssemblyLine.

End of cycle
Updates the System Store with the change log number when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual
Switches off the automatic updating of the System Store with this Connector's state information; instead, you will need to save the state by manually calling the iPlanet Directory Server Changelog Connector's *saveStateKey()* method, somewhere in your AssemblyLine.

Merge Mode
Governs the method used for merging attributes of the Changelog Entry and changed attributes of the actual Directory Entry. The default is **Return only changed data**, and choices are:

Merge changelog and changed data
The Connector merges the attributes of the Changelog Entry with changed attributes of the actual Directory Entry. This option selects the behavior of older versions of Tivoli Directory Integrator and maintains backwards compatibility.

Return only changed data
Returns only the modified or added attributes.

Return both
Returns entry with Changelog Attributes prefixed by "changelog." plus changed attributes of the Directory Entry.

Delivery Mode
Specifies whether to use changelog or (realtime) notifications entries. If the LDAP Server doesn't maintain a changelog, **Realtime** is the only applicable option. The default is **Changelog**.

Use Notifications
Specifies whether to use notification when waiting for new changes in Sun Directory Server. If enabled, the Connector will not sleep or timeout (and corresponding parameters are ignored) but instead wait for a Notification event from the Sun Directory Server.

Batch retrieval
Specifies how searches are performed in the changelog. When unchecked, the Connector will perform incremental lookups (backward compatible mode). When checked, and the server supports "Sort Control", searches will be performed with query "changenumber>=some_value", corresponding to the last retrieval you made. By default, this option is unchecked.

Timeout
Specifies the number of seconds the Connector waits for the next Changelog entry. The default is 0, which means wait forever.

Sleep Interval
Specifies the number of seconds the Connector sleeps between each poll. The default is 60.

Detailed Log
If this field is checked, additional log messages are generated.

Note: Changing Timeout/SleepInterval values will automatically adjust its peer to a valid value after being changed (for example, when timeout is greater than sleep interval the value that was not edited is adjusted to be in line with the other). Adjustment is done when the field editor looses focus.

See also

Standard Changelog in the Sun Directory Server,
Retro Changelog in the Sun Directory Server,
“LDAP Connector” on page 181,
“Active Directory Change Detection Connector” on page 8,,
“IBM Tivoli Directory Server Changelog Connector” on page 123
“z/OS LDAP Changelog Connector” on page 291.

Properties Connector

Tivoli Directory Integrator solutions are packaged into one or more Tivoli Directory Integrator configuration files (XML format) that contain the settings for end point connections, data flow and a host of other features. Although a configuration file can hold everything you need to create a solution, you often may need to use data sources external to the configuration file to modify the behavior of the configuration, such as standard Java properties, Tivoli Directory Integrator external properties and Tivoli Directory Integrator System Store properties.

Property stores are used to hold configuration information in the format of *key=value*. The Properties connector is used to work with such stores, performing operations of reading/writing of properties and encryption/decryption of certain property values. The familiar `global.properties` and `solution.properties` are examples of such property stores.

Individual property stores can be encrypted with individual Certificates, by means of the **Property Key** and **Encrypt** parameters. This allows a certificate that is different from the server certificate to be used for encrypting and decrypting both properties in the file, and the entire file if wanted. This may be useful when multiple developers are working on a project, and credentials cannot be shared.

This Connector uses an internal memory buffer to hold all properties in a properties file. The Connector can also be used to access the JavaVM system properties object.

The Connector supports Iterator, AddOnly, Update, Lookup and Delete mode.

Configuration

The Properties Connector uses the following parameters:

Collection Path/URL
Specifies the properties file to read/write when collection type if File/URL. This parameter is required if the collection type is File/URL.

Create Checkbox, when checked (which is the default), it will automatically create the file. If this checkbox is empty and the file is missing an exception is thrown.

Encrypt
Check to cause this collection of properties to be encrypted using the Password entered. Default value is unchecked, i.e. "false".

Cipher Alg.
The cipher algorithm to use when either **Encryption**=TRUE or the stream contains individually encrypted values. Specify "server" to use Tivoli Directory Integrator server encryption. The Default cipher provided in `global.properties` or `solution.properties` in property `com.ibm.di.server.encryption.transformation`.

When the **Property Key** parameter is specified, this parameter specifies the algorithm to use with that key. If keyalias is not specified, this parameter specifies the algorithm to use when encrypting the entire file. In this case the word "server" means to use Tivoli Directory Integrator server encryption, anything else uses the password from the **Password** parameter as a key for the algorithm.

Password
The secret key to use when encrypting/decrypting the stream/property values.

Only used if **Property Key** is not specified, **Encrypt** is checked, and **Cipher Alg.** is not "server".

Property Key
The name of the Certificate in the server keystore that should be used to encrypt or decrypt individually encrypted values in the Properties File. If the **Encrypt** parameter is set to true, this

certificate will also be used to encrypt or decrypt the entire Properties file. Note that if this parameter is set, it overrides the values of the **Password** parameter.

This parameter is a dropdown list; the dropdown list is automatically filled with the names found in the server keystore.

AutoRewrite
If true, the Connector will write back the contents if any auto-encrypted values were found.

If this parameter is set to true, the collection will immediately be written back if any value was automatically encrypted. If a property is marked with "{protect}-" in front of the property name, the value will be automatically encrypted if it is not encrypted. If this parameter is not set to true, the collection must be written back by programmatic means.

Detailed Log
Checking this box will cause additional log messages to be generated.

Using the Connector

The Property Connector is used to connect to standard .property files, Java Properties or the System Store User Property Store. It provides encryption/decryption of the stores being read/written.

The typical behavior of this connector is to connect to a .property file specified by its URL. This can be achieved by setting the **collection** parameter of the connector, and constitutes "User-Defined" properties.

However, you can also access the system-defined property stores: JVM ("java") properties, User Property Store, `global.properties` and `solution.properties`. In order to do this, you need to set the **collectionType** property of the connector. It is not exposed in the configuration screen but can be set with the following script (for example, put in the Prolog>Before Initialize hook):

- **JVM Properties**: `thisConnector.connector.setParam("collectionType", "Java-Properties");`
- **User Property Store**: `thisConnector.connector.setParam("collectionType", "System-Properties");`
- **global.properties**: `thisConnector.connector.setParam("collectionType", "Global-Properties");`
- **solution.properties**: `thisConnector.connector.setParam("collectionType", "Solution-Properties");`

Note: These property collections are those that show up in the "Properties" folder in the Config Browser for a given configuration file. These can be modified using the Config Editor, and this may make it unneccesary to use this Connector to access or alter any of the properties in these property collections at runtime.

All of these stores are shared within the same JavaVM, which means that an AssemblyLine writing to the System Store will affect all other AssemblyLines in the JavaVM reading from the same store.

All of the properties in the global and solution stores are propagated to the Java property store by the Tivoli Directory Integrator server at startup in that order. The point to make here is that the global and solution stores can now be discretely addressed and modifications to these files, if permitted, can also be made. Each property store is given a unique name that is unique within the confines of a configuration instance. If a Tivoli Directory Integrator server runs multiple configuration instances, they will share the Java, global, solution and all System-Store property stores (for example, system) but all others are local to the configuration instance.

Note: When using the Connector to deal with external properties, the **Auto-Rewrite** parameter should be set to true if you want to automatically write back encrypted properties without calling an explicit "commit".

Properties File Format

```
# comment
' comment
// comment
!include filename
!merge filename

[{protect}-]keyword <colon | equals> [{encr}]value
```

Notes:

1. The optional {protect}- prefix signals that the value either is or should be encrypted. When the value starts with the character sequence {encr} it means that the value is already encrypted.
2. "!include" reads an external file/URL with properties which are written unconditionally to the current property map.
3. "!merge" reads an external file/URL with properties which are written to the current property map if the property does not already exist (non-destructive write).
4. TDI currently uses the equal sign "=" or colon ":" as the separator in key/value pairs property files, whichever is first. Using equal signs or colons in property names and property values is therefore not supported. The property file key/value separator in TDI V6.0 and earlier was only the ":" character; therefore, property files migrated from V6.0 and earlier may require editing.

Syntax checking is used on properties files that are read in directly by the Properties Connector, the Tivoli Directory Integrator Server and the CE. If any nonblank line does not adhere to the properties file format, an Exception will be thrown.

Headers in the Property file:

The first one or two lines in a Property File will be lines beginning with this String

```
##{PropertiesConnector}
```

This signifies that this line is a header that is rewritten every time the Property File is written.

The first line will look like this

```
##{PropertiesConnector} savedBy=user, saveDate=date
```

where `user` is the name of the user that saved the file and date is the date the file was saved.

If the **Property Key** parameter was specified when writing the file, the next line will look like this

```
##{PropertiesConnector} encryptionKey=keyAlias
```

where keyAlias is the value of the **Property Key** parameter.

See also

"Property Store" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.

Server Notifications Connector

The Server Notifications Connector is an interface to the IBM Tivoli Directory Integrator (Tivoli Directory Integrator) notification system. It listens for and reports, as well as issues, Server API notifications. The Connector provides the ability to monitor various processes taking place in the Tivoli Directory Integrator Server, such as AssemblyLine stop and start process events, as well as issue custom server notifications.

The Server Notifications Connector supports the Iterator and AddOnly modes.

Iterator Mode:

Depending on how it is configured the Server Notifications Connector in Iterator mode is capable of listening to and reporting either local or remote Server API notifications, but not both during the same Connector session.

The "Local" connection type should be used when the Connector is run in the same JVM as the Tivoli Directory Integrator Server which sends notifications.

The "Remote" connection type should be used when the Connector connects to a remote Tivoli Directory Integrator Server run in a different JVM.

AddOnly Mode:

The Connector in AddOnly mode sends Server API custom (that is, user-defined) notifications through either the local or the remote Server API session, but not both during the same Connector session.

The "Local" connection type should be used when the Connector is run in the same JVM as the Tivoli Directory Integrator Server which sends notifications.

The "Remote" connection type should be used when the Connector connects to a remote Tivoli Directory Integrator Server run in a different JVM.

The data needed for creating the notification objects is retrieved from the *conn* Entry passed to the Connector by the AssemblyLine. The Connector looks for fixed-name Attributes in this Entry, retrieves their values, builds the notification object using these values and emits this notification object through the Server API. For more information about the fixed-name Attributes please see the "Schema" section.

Since each Server API notification also causes a corresponding JMX notification to be emitted, the Server Notifications Connector in AddOnly mode also indirectly sends JMX notifications. For more information about the details of custom notifications please see section “Schema” on page 225.

Encryption

The Server Notifications Connector provides the option to use Secure Sockets Layer (SSL) when the connection type is set to **remote**. If the remote Tivoli Directory Integrator server accepts SSL connections only, the Server Notifications Connector automatically establishes an SSL connection provided that a trust store on the local Tivoli Directory Integrator Server is configured properly. When SSL is used, the Connector uses a Server API SSL session, which runs RMI over SSL.

Trust store

A trust store on the local Tivoli Directory Integrator Server is needed because when the remote Tivoli Directory Integrator Server fires a notification a new SSL connection to the local Tivoli Directory Integrator Server is created and in order for this new SSL connection session to be established the local Tivoli Directory Integrator Server must trust (through its trust store) the remote Tivoli Directory Integrator Server SSL certificate. A trust store is configured by setting the appropriate values for the

javax.net.ssl.trustStore, javax.net.ssl.trustStorePassword and javax.net.ssl.trustStoreType properties in the global.properties or solution.properties files.

Authentication

SSL Authentication

The Server Notifications Connector is capable of authenticating by using a client SSL certificate. This is only possible when the remote Tivoli Directory Integrator Server API is configured to use SSL and to require clients to possess SSL client certificates. A trust store must be configured properly on the local Tivoli Directory Integrator server.

Username and Password Authentication

The Server Notifications Connector is capable of using the Server API username and password authentication mechanism. The desired username and password can be set as a Connector parameter, in which case the Connector will use the Server API username and password authentication mechanism. If SSL is used and a username and password have been supplied as Connector parameters, then the Connector will use the supplied username and password and not an SSL client certificate to authenticate to the remote Tivoli Directory Integrator Server.

Configuration

The Server Notifications Connector uses the following parameters:

Connection Type
: Determines whether the Server Notifications Connector will listen for and emit local or remote Server API notifications. The available values for this parameter are remote and local. local means that the Connector will only listen for and notifications in the local Java Virtual Machine. remote means that the Connector will connect to a remote Tivoli Directory Integrator Server system and register for and emit notifications in the Java Virtual Machine of that remote system.

RMI URL
: Specifies the Remote Method Invocation (RMI) URL used to connect to the remote Tivoli Directory Integrator Server system. This parameter is only taken into account if the connectionType parameter is set to remote. An example value for this parameter is:

```
rmi://127.0.0.1:1099/SessionFactory
```

Username
: Specifies the user name the Connector uses to authenticate to the Tivoli Directory Integrator server. This parameter is only taken into account if the **Connection Type** parameter is set to *remote*.

Password
: Specifies the password the Connector uses to authenticate to the Tivoli Directory Integrator server. This parameter is only taken into account if the **Connection Type** parameter is set to *remote*.

Filter Config Instance ID
: Specifies a Config Instance ID, which the Connector will use to filter event notifications. If this parameter is specified, the Connector will only report notifications that have this Config Instance ID. This parameter is only taken into account if the Connector mode is Iterator.

Filter Notification ID
: Specifies a Notification ID, which the Connector will use to filter event notifications. If this parameter is specified the Connector will only report notifications which have the specified notification ID. This parameter is only taken into account if the Connector mode is Iterator.

Timeout (seconds)
: Specifies the maximum number of seconds to wait for a notification. After this timeout expires, the Connector will terminate. If this parameter value is set to "0", then the Connector will wait forever. This parameter is only taken into account if the Connector mode is Iterator.

Receive All Server API Events
Specifies if "di.*" notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

Receive All Config Instance Events
Specifies if "di.ci.*" notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

Receive Config Instance Start Events
Specifies if "di.ci.start" notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

Receive Config Instance Stop Events
Specifies if "di.ci.stop" notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

Receive Configuration Updated Events
Specifies if "di.ci.file.updated" notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

Receive All AssemblyLine Events
Specifies if "di.al.*" notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

Receive AssemblyLine Start Events
Specifies if "di.al.start" notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

Receive AssemblyLine Stop Events
Specifies if "di.al.stop" notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

Receive Server Shutdown Event
Specifies if "di.server.stop" notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

Use Custom Notification
Specifies whether the Connector will receive any additional or custom notifications. If it is checked the additional/custom notifications can be specified in the **Custom Notification types** Connector parameter. This parameter is only taken into account if the Connector mode is Iterator.

Custom Notification types
Specifies the notification types of additional or custom Server API notifications which the Server Notifications Connector will listen to and report. Each notification type must be typed on a separate line. This parameter takes effect only if the **Use Custom Notification** parameter is true, and is only taken into account if the Connector mode is Iterator.

Debug
Turns on debug messages. This parameter is globally defined for all Tivoli Directory Integrator components.

Schema

Iterator mode

The Server Notifications Connector in Iterator mode sets the following Attributes in the Input Attribute Map:

event.rawNotification
The notification event object (com.ibm.di.api.DIEvent). It contains the complete information about the Server API event generated in the core of the Tivoli Directory Integrator Server.

event.type
The type of the notification event (java.lang.String). This Attribute specifies what has happened. An example value of this Attribute would be "di.al.start".

event.id
The notification ID (java.lang. String). This Attribute specifies the source of the event, that is, which Tivoli Directory Integrator component has fired this event. An example value of this Attribute would be "AssemblyLines/tcp".

event.userData
The notification user data. (java.lang.Object). Optional user-defined information whose purpose is to convey more information related to the event, for example why the event happened, where the event happened, etc. This Attribute is only available if such user-data was actually passed on generating the event.

For AssemblyLine events this Attribute contains the AssemblyLine unique code, which can be used to unambiguously identify the AssemblyLine instance which has generated this event (for example "1709375019").

If the userData object is of type com.ibm.di.entry.Entry, then its Attributes are also mapped in the generated output Entry, so they can be directly accessed in the Assembly Line (for example, `conn.getAttribute("event.userData.hostname")`) without a need of additional scripting in order to make them available.

event.configInstanceId
The config instance ID (java.lang.String). The ID of the loaded and running config instance which has fired this event. An example value of this Attribute would be "C__dev_assembly_TCPServer.xml".

event.dateCreated
The date object stores the time and date when this notification was created. (java.util.Date).

AddOnly mode

The Server Notifications Connector in AddOnly mode expects to receive the following Attributes from the Output Attribute Map:

event.type
The type of the notification event (java.lang.String). This Attribute specifies what event the custom notifications signals. Since this is a user-defined event this can be any String. An example value of this Attribute would be "myAL.DBRecord.Committed". This Attribute's presence in the conn Entry is required. If this Attribute is missing from the conn Entry, then the Connector throws an Exception.

The value supplied by the user for this Attribute will be prefixed with the "user." prefix by the Connector when building the notification object. For example if the type passed by the user is "process.X.completed" the type of the event broadcasted will be "user.process.X.completed".

event.id
The notification ID (java.lang. String). This Attribute specifies the source of the event, that is, which Tivoli Directory Integrator component has fired this event. Since this is a user-defined event this can be any string, for example "myAssemblyLine_5". This Attribute's presence in the conn Entry is required. If this Attribute is missing from the conn Entry, then the Connector throws an Exception.

event.userData
The notification user data. (java.lang.Object). Optional user-defined information whose purpose is to convey more information related to the event, for example why the event happened, where the event happened, etc. The presence of this Attribute in the *conn* Entry is optional.

System Queue Connector

Introduction

The System Queue provides a subsystem similar to Java Message Service (JMS) for IBM Tivoli Directory Integrator. It is designed for storing and forwarding general messages and Tivoli Directory Integrator Entry Objects, between Tivoli Directory Integrator Servers and AssemblyLines.

The System Queue Connector is the mechanism for AssemblyLines to interface with the System Queue. To learn more about the System Queue and its configuration, refer to the System Queue section in the *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*.

The System Queue Connector can be used with AssemblyLines in Iterator and AddOnly modes:

- In Iterator mode, the Connector retrieves Tivoli Directory Integrator Entry objects from a specified message queue.
- In AddOnly mode, the Connector stores Tivoli Directory Integrator Entry objects in the specified message queue.

Note: If two JMS clients retrieve messages from the same JMS queue simultaneously, an error might occur. Avoid solutions which use several instances of the System Queue Connector retrieving messages from the same JMS queue simultaneously. However, an instance of the System Queue Connector writing to a queue and another instance of the Connector reading from that same queue at the same time is acceptable.

The System Queue Connector uses the Server API to access the System Queue. The Connector uses both the local and remote interfaces of the Server API, allowing the Connector to operate on a Tivoli Directory Integrator System Queue running on a remote computer. The Connector's ability to operate on a remote computer, coupled with the System Queue's capability to connect to remote JMS servers, results in the ability to use some quite complex deployment scenarios. For example: a Tivoli Directory Integrator server and a System Queue Connector deployed on machine A, working through the remote Server API with the Tivoli Directory Integrator server and System Queue on machine B, which in turn interface with a JMS server deployed on machine C.

In Tivoli Directory Integrator 7.1, the System Queue is enabled by default by the install process and the MQe Queue Manger is set up with two queues:

- **_default** – serves as a general purposes queue
- **passwords** – this queue is used by the JMS Password Store components for storage of password changes.

Configuration

The System Queue Connector uses the following parameters:

Connection Type

This parameter determines whether the System Queue Connector works with the System Queue of the local Tivoli Directory Integrator server or with the System Queue of a remote Tivoli Directory Integrator server. The available values for this parameter are *local* and *remote*.

- The value *local* specifies that the Connector will use the local Server API interfaces and will work with the System Queue of the local Tivoli Directory Integrator server. This is the default.

  The value *remote* specifies that the Connector will use the remote Server API interfaces and will work with the System Queue of a remote Tivoli Directory Integrator server. In that case, the RMI URL parameter is required, the Connector configuration (both the Connector parameters and the SSL configuration of the local Tivoli Directory Integrator Server) must match the configuration of the remote Tivoli Directory Integrator Server.

RMI URL
The Remote Method Invocation (RMI) URL used to connect to the remote Tivoli Directory Integrator Server system. An example value (and default) for this parameter is:

```
rmi://127.0.0.1:1099/SessionFactory
```

This parameter is taken into account only if the `connectionType` parameter is set to `remote`.

Username
Used to authenticate to the remote Tivoli Directory Integrator server using the user name and password authentication mechanism of the Server API. This parameter is taken into account only if the `connectionType` parameter is set to `remote`.

Password
Used to authenticate to the remote Tivoli Directory Integrator server. This parameter is taken into account only if the `connectionType` parameter is set to *remote*.

Queue Name
Specifies the name of the JMS queue with which the Connector will work. In Iterator mode, the Connector retrieves Entry objects from this queue. In Add-Only mode, the Connector stores Entry objects in this queue.

Timeout
Specifies the amount of time in seconds the Connector will wait before returning a null Entry object. If a value of zero (0) is specified for this parameter, the Connector will immediately return if there are no available Entry objects in the queue. If a negative value is specified for this parameter then the Connector will wait indefinitely or until an Entry object becomes available in the queue. The default value is -1.

Detailed Log
This parameter turns on debug messages This parameter is globally defined for all Tivoli Directory Integrator components

Security, Authentication and Authorization

Encryption

When the connection type is set to *remote* and the remote Tivoli Directory Integrator server is configured to use Secure Sockets Layer (SSL), then the System Queue Connector uses SSL, provided that a trust store on the local Tivoli Directory Integrator Server is configured properly. When SSL is used, the Connector uses a Server API SSL session, which runs RMI over SSL.

Note: Of the standard JMS Drivers only the driver for MQ supports SSL out of the box. The MQe JMS Driver only works with a local Queue Manager – this is mandated by the MQe architecture. The JMS Script Driver is a generic driver which supports whatever the corresponding user-provided Javascript supports.

Authentication

Username and password authentication: The System Queue Connector can use the remote Server API username and password authentication. The Connector does not implement any authentication itself. The username and password supplied to the Server API are configured as Connector configuration parameters.

SSL certificate-based authentication: The System Queue Connector is capable of authenticating by using a client SSL certificate. This is only possible when the remote Tivoli Directory Integrator Server API is configured to use SSL and to require clients to possess SSL client certificates. A trust store must be configured properly on the local Tivoli Directory Integrator server.

If SSL is used and a user name and password have been supplied as Connector parameters then the Connector will use the supplied user name and password and not the SSL client certificate to authenticate to the remote Tivoli Directory Integrator Server.

Authorization

The Server API authorization mechanism is applied to the Server API session the System Queue Connector establishes to the Tivoli Directory Integrator Server. With the Tivoli Directory Integrator 7.1 Server API once the System Queue Connector is authenticated it can use the Tivoli Directory Integrator System Queue.

See also

"JMS Connector" on page 151,

"System Queue" in *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*,

Usage example SystemQueue_SonicMQ_example.xml using Sonic MQ in *TDI_install_dir*/examples/SonicMQ,

Usage example SystemQConn_jmsScriptDriver_example.xml using the WebSphere Default JMS Provider in *TDI_install_dir*/examples/was_jms_ScriptDriver.

Windows Users and Groups Connector

The Windows Users and Groups Connector (in older versions of Tivoli Directory Integrator this was called the NT4 Connector) operates with the Windows NT® security database. It deals with Windows users and groups (the two basic entities of the Windows NT security database). This Connector can both read and modify Windows NT security database on the local Windows machine, the Primary Domain Controller machine and the Primary Domain Controller machine of another domain.

Note: This Connector is dependent on a Windows NT API, and only works on the Windows platform.

The Connector is designed to connect to the Windows NT4 and Windows 2000 SAM databases through the Win32 API for Windows NT and Windows 2000/2003 user and group accounts. You can connect to a Windows 2000 SAM database, but the Connector only reads or writes attributes that are backward-compatible with NT4 (in other words, the Windows Users and Groups Connector has a predefined and static attribute map table consisting of NT4 attributes). Windows 2000/2003 native attributes or user-defined attributes are therefore not supported by this Connector.

See “Windows Users and Groups Connector functional specifications and software requirements” on page 234 for a full functional specification of the Connector, architecture description as well as hardware and software requirements.

Note: This component is not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.

Preconditions

To successfully run the Windows Users and Groups Connector and obtain all of its functionality, the Connector must be run in a process owned by a user who is a member of the local Administrators group, and have logon privileges to the domain controller and other domains (if accessed). This precondition can be omitted if the **UserName** and **Password** parameters of the Connector are set to specify an account with the requirements stated above.

The Windows Users and Groups Connector is designed and implemented to work in the following modes:

- Iterator
- Lookup
- AddOnly
- Delete
- Update

Note: This Connector does not support Advanced Link Criteria (see "Advanced link criteria" in *IBM Tivoli Directory Integrator V7.1 Users Guide*).

Configuration

The Connector needs the following parameters:

Computer Name
: The name of the machine (for example, **ntserver01**) or its IP address (for example, **212.52.2.218**) where the Connector operates. The machine IBM Tivoli Directory Integrator is running on must be in the same Domain or Workgroup as the target system.

Username
: If blank, no logon to the specified machine is performed and the Connector has the privileges of the process in which IBM Tivoli Directory Integrator is run. If some value is set, then the Connector attempts to log on to the **Computer Name** machine with this user name and the password specified by the **Password** parameter.

Password
The value of this parameter is taken into account only when the parameter **Username** is set with a non-blank value. It then specifies the password used for the logon operations.

Entry Type
Must be set to **User** (specifying that the Connector operates with data structured by Users) or **Group** (specifying that the Connector operates with data structured by Groups).

Page Size
Specifies the number of Entries (Users and Global Groups) that Windows NT or Active Directory return in one chunk when the Connector retrieves Users and Global Groups. Must be a number between **1** and **100**.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Constructing Link Criteria

Construct link criteria when using the Windows Users and Groups Connector in Lookup, Update and Delete modes. The Connector supports Link Criteria that uniquely identifies one entry only. The format is strict, and passing a Link Criteria that doesn't match this format results in the following exception:

```
Unsupported Link Criteria structure.
```

The following is the Link Criteria structure that must be used, depending on Entry Type:

User Windows Users and Groups Connector **Entry Type** parameter is set to **User**. This parameter consists of just one row where:

- Connector attribute is set to **UserName**.
- Operand is set to **equals**.
- Value is set to a name of a user account (for example, **user name**) or configured by a template to receive the name of a user account.

Group Windows Users and Groups Connector **Entry Type** parameter is set to **Group**. This parameter consists of two rows as follows:

1. Initial row:
   - Connector attribute is set to **GroupName**.
   - Operand is set to **equals**.
   - Value is set to a name of a group account (for example, **group name**) or configured by a template to receive the name of a group account.
2. Second Row:
   - Windows Users and Groups Connector attribute is set to **IsGlobal**.
   - Operand is set to **equals**.
   - Value is set to **True** to indicate that the group account specified in the first row is global, or **False** to indicate that the group account is local. Can also be configured by a template to receive **True** or **False** values indicating global or local group accounts.

Other

User and Group account names:

On Domain Controller Machine
Users and groups are retrieved and must be accessed in the following formats:

USER_NAME,*GROUP_NAME*

On Non-Domain Controller Machine
Local users and groups are retrieved and must be accessed in the following format:

USER_NAME,*GROUP_NAME*

Global groups and domain users (can be members of a local group on a non-domain controller machine) are retrieved and must be accessed in the following format:

DOMAIN_NAME\GLOBAL_GROUP_NAME,*DOMAIN_NAME\USER_NAME*

Creating a new user: When creating a new user with the Windows Users and Groups Connector, if any of the following attributes are omitted or assigned a **null** value, they are automatically assigned a default value as follows:

Flags The account is marked as **normal account** and **user password never expires**.

AccountExpDate
A value that indicates that the **account never expires** is set.

LogonHours
A value that indicates that there is no time restriction set (for example, the user can log on always).

Setting user password

Remember that a user password value cannot be retrieved. Windows stores this in a format that cannot be read. If an AssemblyLine copies users from one Windows machine to another, you must set the **Password** attribute value manually.

When adding a user, passing the **Password** attribute with no value results in creating a user with an empty password.

When modifying a user, passing the **Password** attribute with no value results in retaining the old password.

Setting user Primary Group/global groups membership

All Domain Users must be members of their Primary Groups. This means that the value set in the **PrimaryGroup** attribute must be present in the **GlobalGroups** attribute. If there is no value for the PrimaryGroup attribute then it will be set to "Domain Users".

Operating with groups

There are certain groups that are predefined and special for Windows, and there are certain operations that are not enabled on these groups. Such operations are delete, rename and modification of some of their attributes. Any attempt to try such an invalid operation over any of these groups results in an exception thrown.

Here is the list of these groups, structured by Windows installations:

Domain Controller:

- Global groups
  - Domain admins
  - Domain users
- Local groups
  - Administrators
  - Users
  - Guests
  - Backup operators
  - Replicator
  - Account operators
  - Print operators
  - Server operators

Non-Domain Controller:

- Local groups
  - Administrators
  - Users
  - Guests
  - Backup operators
  - Replicator
  - Power Users

Character sets

Unicode is supported.

Examples

Navigate to the *root_directory*/examples/NT4 directory of your IBM Tivoli Directory Integrator installation.

Windows Users and Groups Connector functional specifications and software requirements

The Windows Users and Groups Connector implements Windows Users and Groups database access for both user and group management on Windows systems according to Windows definitions and restrictions as outlined below. For additional background information, see Overview of Users and Groups and Managing local and remote Users and Groups.

Functionality

Extract user and group data: The Windows Users and Groups Connector reads both user and group information from the Windows Users and Groups repository, including group and user metadata as well as relationship information (for example, **users** group and **groups** group membership). The Connector reads both local and domain user or group data. Data is read from Windows, then organized and provided in the containers expected by IBM Tivoli Directory Integrator.

Add user and group data: The Windows Users and Groups Connector adds user information to both local machines and domain controllers, and it adds group information to both local machines and domain controllers. When operating with a domain controller, the Connector can create both local and global groups. When operating with a machine that is not a domain controller, the Connector can only create local groups, according to security restrictions set by Windows.

Modify group membership: The Windows Users and Groups Connector modifies group membership for both local and global groups. In accordance with Windows NT security restrictions, members can be assigned to groups as follows:

- A global group can have users from its domain as members only.
- A local group can have global groups and users from its domain or any trusted domain as members. However, a local group cannot contain other local groups.
- Users on a local machine can exist without being members of a group.
- Each user on a domain controller must belong to a Primary Group. The Primary Group for a user can be any global group in the domain. While the user's Primary Group can be changed, he is always a member of his Primary Group.

Modify user and group data: The Windows Users and Groups Connector modifies external and group properties on both local machines and domain controllers. When connected to a domain controller, the Connector is able to modify the properties of both local and global groups.

Delete user and group data: The Windows Users and Groups Connector can remove users from both local machines and domain controllers, and it can remove local groups from both local machines and domain controllers. When operating with a domain controller, the Connector can remove both local and global groups.

System Store Connector

The System Store Connector provides access to the underlying System Store. The primary use of the System Store Connector is to store **Entry** objects into the System Store tables. However, you can also use the connector to connect to an external Derby, DB2 8.1, Oracle, Microsoft SQL*Server or IBM SolidDB database, not just the database configured as the System Store. Each **Entry** object is identified by a unique value called the key attribute.

The System Store Connector creates a new table in a specified database if one does not already exist. If you iterate on a non-existing table, the (empty) table is created, and the Iterator returns no values.

The System Store Connector uses the following SQL statements to create a table and set the primary key constraint on the table (Derby syntax):

```
"CREATE TABLE {0} (ID VARCHAR(VARCHAR_LENGTH) NOT NULL, ENTRY BLOB );
ALTER TABLE {0} ADD CONSTRAINT {0}_PRIMARY Primary Key (ID);"
```

This connector provides pre-set SQL statements for a number of popular databases, but there is also the ability to modify them as you see fit. For other databases, you must enter your own, equivalent SQL statements (multiple ones, if required) by specifying these in the **Create Table Statement** parameter. The parameter can not be empty; in that case, an exception is thrown.

Notes:

1. The VARCHAR_LENGTH value is picked up from the `com.ibm.di.store.varchar.length` property set in the Properties Store (TDI-P). The default VARCHAR_LENGTH is set to 512. You can change this value by setting the value of the `com.ibm.di.store.varchar.length` in the Properties Store.
2. Another attribute, `tdi.pesconnector.return.wrapped.entry`, exists for TDI 6.0 backward compatibility. If you define this property in the Tivoli Directory Integrator `global.properties` file and set it to *true* , then Tivoli Directory Integrator reverts back to its earlier behavior where for example, the `findEntry()` method (used by the system in Iterator, Lookup and Update modes) would return an Entry object of the format: [ENTRY: <Instance of Entry object containing Attributes passed by user>]. In TDI 6.0, in order to obtain the original passed attributes, you would need to write JavaScript code something like this:

   ```
   Entry e = (Entry)conn.getAttribute("ENTRY");
   ```

   at some appropriate place, after which *e* contains the Attributes originally passed in when writing to the System Store. You could do this in the Input Attribute Map Hook where you would have to carefully map the Attributes in *e* to the *work* entry, or use a Script Component after this Connector to unpack the composite *entry* attribute in the *work* entry using the aforementioned JavaScript example (substitute *work* for conn.)

   In Tivoli Directory Integrator 7.1, by default the entry is unwrapped and therefore all attributes passed by you are now directly available as attributes in the Entry. The above scripting will not be needed any longer (unless you set the `tdi.pesconnector.return.wrapped.entry` attribute to *true*.)
3. The System Store Connector operates in the following modes: AddOnly, Update, Delete, Iterate, Lookup. However, AddOnly, Update and Delete operations are not permitted on the Delta Tables and Property store tables.

The Connector supports both simple and advanced Link Criteria.

This Connector, like the JDBC Connector it is based upon, in principle can handle secure connections using the SSL protocol. However, it may require driver specific configuration steps in order to set up the SSL support. Refer to the manufacturer's driver documentation for details.

Configuration

The System Store Connector requires the following parameters.

Database
The location of the database. This is an optional parameter; if left blank, the System Store as configured in property `com.ibm.di.store.database` in the `global.properties` file is used. Note that this is the value displayed in the **Store -> View System Store** screen.

Username
The name of the user used to make a JDBC connection to the specified database. Only the tables available to this user are shown. If this is not specified then the value of the `com.ibm.di.store.jdbc.user` property set in the `global.properties` file is used as the default value.

Password
The password of the user used to make a JDBC connection to the specified database. If this is not specified then the value of the `com.ibm.di.store.jdbc.password` property set in the `global.properties` file is used as the default value.

Key Attribute Name
The attribute name giving the unique value for the entry. This is a required parameter.

Note: You can specify multiple Key Attribute Names separated by the "+" sign. The System Store Connector will concatenate these into a single `varchar(255)` key to obtain a unique key.

Selection Mode
Specify **All**, **Existing** or **Deleted**. In order to use the **Existing** and **Deleted** keywords, the Connector must reference a Delta table in the System Store. When Delta is enabled on an Iterator, the AssemblyLine stores a sequence property in the database, adding a sequence number to each entry read from the source. This parameter is to be used on Delta tables only.

Note: Delta table names in Tivoli Directory Integrator 6.0 and above, have an "IDI_DS_" prefix added to the *identifier* specified in "Delta Store" field of the Delta configuration tab.

Table Name
The table name to store the entries in. This is a required parameter. The System Store Connector will create a table with the specified table name if it does not exist.

Notes:

1. The "Select" button in the Connector configuration tab of the connector provides a list of tables in the connected database. Only the tables available to the user specified in the Username field are shown.
2. The "Delete" button in the Connector configuration tab can be used to delete a selected table. Ideally, the Delete button should be used when an AL has run and you would now want to delete the table created by the System Store Connector. This does not work with the Delta tables.
3. The table name must be a valid name for the database you are accessing. In most cases, this will mean the name must begin with a letter, and otherwise may only contain letters, digits and the underscore (_) character.

JDBC Driver
This parameter contains the Java class name of the JDBC driver (instead of the database name as in previous versions. Existing configurations will be migrated automatically).

If the parameter is left empty or one of its provided options is selected, the Create Table Statement parameter is initialized with a default "CREATE TABLE" statement for the database used. If **JDBC Driver** is not specified the JDBC driver configured in the System Store settings is used to obtain the proper value for **Create Table Statement**.

This parameter enables the System Store Connector to connect to different System Store databases without changing the System Store settings.

The possible values are:

- org.apache.derby.jdbc.ClientDriver
- org.apache.derby.jdbc.EmbeddedDriver
- com.ibm.db2.jcc.DB2Driver
- oracle.jdbc.OracleDriver
- com.microsoft.sqlserver.jdbc.SQLServerDriver
- solid.jdbc.SolidDriver

The default value is empty.

Create Table Statement
The "CREATE TABLE" SQL statement used to create the tables in the selected data source. You are required to enter the correct CREATE TABLE statement corresponding to the database that you choose to connect to. Otherwise the Connector will fail to create the table if the table is missing.

Delete table on close
If this value is set to **true** then the table created by the System Store Connector will be dropped when the Connector terminates.

SQL Select
The select statement to execute when selecting entries for iteration. Specifies the WHERE clause. This will be used as a search filter to return the data set in Iterator mode. If this is left blank, the default construct (`SELECT * FROM TABLE`) is used, where TABLE is the name specified in the "Table Name" field.

Commit
Controls when database transactions are committed. Options are:

- **After every database operation**
- **On Connector Close**
- **Manual**
- **End of Cycle**

Manual means user must call the *commit()* method of the System Store Connector — or, alternatively, *rollback()* if your logic requires this.

Detailed Log
If this field is checked, additional log messages are generated.

Using the Connector

The System Store Connector provides access to the tables created in the System Store. The System Store can be located on any DB server for which a JDBC driver is available. Furthermore, if the System Store uses Derby, it can be configured to run in either embedded (inside the Tivoli Directory Integrator Server process) or networked mode. The connector is able to resolve globally defined parameters to obtain a connection the default System Store. In order to configure a connection to a different DB at least the following parameters must be explicitly provided: **Database**, **Username**, **Password** or **JDBC Driver**.

The correct way to specify the database and JDBC Driver for different configurations of System Store is given below.

Note: The examples are specific to Windows.

Using System Store Connector with embedded Derby server configured as System Store
Database: f:\Program Files\IBM\IBMDirectoryIntegrator/Derby
JDBC Driver: org.apache.derby.jdbc.EmbeddedDriver

Note: In the embedded mode of operation, the Derby server is automatically started and the specified database is booted into the database if it exists. If it does not exist a new database is created at the specified location.

Using System Store Connector with networked Derby server configured as System Store

Database: jdbc:derby://localhost:1527/E:\TDI\TDISysStore;create=true
JDBC Driver: org.apache.derby.jdbc.ClientDriver

Notes:

1. It is important to specify the "create=true" flag in the database URL. This will create the database if it does not exist. This is required when Derby is configured to run in networked mode.
2. In networked mode of operation, the Derby server my need to be started manually. For details regarding the ways in which a Derby server can be started in networked mode, please refer to the chapter on *System Store* in *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*

Using System Store Connector with DB2 8.1 server as System Store

Database: jdbc:db2://machine-name:50000/testDB
JDBC Driver: com.ibm.db2.jcc.DB2Driver

Notes:

1. The DB2 instance and the DB2 database must be created ahead of time for it to be used as System Store.
2. The specified instance must be running on the specified port in the database URL.

See also

Connector usage examples in *TDI_install_dir*/examples/systore, and *TDI_install_dir*/examples/SystemStore; The chapter on *System Store* in *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*.

RAC Connector

Introduction

"RAC" stands for Remote Agent Controller, however the current name for this technology is *Agent Controller*.

The Agent Controller is a server that enables client applications to interact with agents under its domain: http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.tptp.platform.agentcontroller.doc.user/tasks/rac/tworkwac.html

A Generic Log Adapter (GLA) transforms proprietary log and trace data to the Common Base Event format (http://www.ibm.com/developerworks/library/specification/ws-cbe/). The rationale for a Generic Log Adapter is that reading log files is messy and making parsers for all types of logs is not scalable and one tends to customize anyway. A GLA can act as an agent of an Agent Controller so that clients can monitor remote application logs.

More information about Agent Controller and Generic Log Adapter can be found on http://www.eclipse.org/tptp/home/documents/index.php.

The RAC Connector can read data from and write data to RAC:

- In **AddOnly** mode the RAC Connector publishes Common Base Events through a Logging Agent. In this mode, the RAC Connector uses an instance of the "CBE Function Component" on page 395 to help convert the input schema attributes into a single Common Base Event object.

  It registers as an agent within the local Agent Controller and sends it the Common Base Event objects, which it receives from the AssemblyLine.

  The RAC Connector does not require the local Agent Controller to be running at the time it is initializing. As soon as the local Agent Controller is launched, the Logging Agent is registered and gets ready to be monitored by clients.

  The important point is that the Connector will not report an error if the local Agent Controller is not active.

  The Connector will not complain even if there is no Agent Controller installation on the local machine. Of course, no Logging Agent will be registered then.
- In **Iterator** mode the RAC Connector acts as a client of a remote Logging Agent.

  As such, it contacts the Agent Controller on the remote machine, obtains a handle to a certain Logging Agent and starts receiving log data in the form of Common Base Event objects.

  If the remote Agent Controller goes down the Connector hangs waiting for a response from the Agent Controller (this is due to the current client library realization). – thus any reconnect logic cannot be used.

  The Connector uses an internal queue to store the incoming Common Base Events (CBEs). As a result the Connector can keep fetching CBEs even after the Agent Controller has gone down because the queue could still have events in it. Due to restrictions caused by the Agent Controller client library, the Connector can not process Common Base Event objects, which when serialized as XML are larger than 8Kb. If a data portion larger than 8Kb arrives, the Connector will not process any more events and will wait until the remote Logging Agent dies. This is a limitation of the client library implementation.

  On termination, the Connector detaches from the remote agent. If this procedure is skipped for some reason (for example the JVM is killed), no other client will be able to monitor the agent. Moreover, the agent will still think that it is being monitored.

  For example, if one runs the Connector from the Config Editor and manually stops the AssemblyLine, the Connector will not have a chance to detach from the Logging Agent. So if the Connector is run again, it will not receive any data from the agent, because the Agent Controller (and the agent) thinks that the agent is already being monitored.

Only one client can monitor a Logging Agent at a given time, so no two RAC Connectors in Iterator mode should be pointed at the same agent at the same time.

Configuration

The Connector's title, shown in the Configuration panel is "RAC Connector". Parameters are:

Remote Logging Agent Name
Used in Iterator Mode.

The name of the remote Logging Agent to be monitored.

Agent Controller Host
Used in Iterator Mode.

Host of the remote Agent Controller. Default value is "localhost".

Agent Controller Port
Used in Iterator Mode.

Port of the remote Agent Controller. Default value is 10006.

Receiving Queue Size
Used in Iterator Mode.

The size of the queue, where the received events are buffered before the Connector manages to read them. Default value is 1024.

Wait For Dead Agent's Data
Used in Iterator Mode.

Timeout (in seconds) for each data reception after the remote agent dies. If this timeout expires, the agent's data is considered depleted and the Connector terminates. Default value is 5.

Connection Timeout
The Socket timeout (in seconds) for the connection to the Agent Controller. Default value is 5.

Logging Agent Name
Used in AddOnly Mode.

The name of the Logging Agent within the local Agent Controller. Default value is "tdi_logging_agent".

Wait to be monitored
Used in AddOnly Mode.

Time to wait (in seconds) for the agent to be monitored before data is sent to RAC. If zero, waits forever. Default value is 0.

Detailed log
When checked, additional log messages will be generated.

Using the Connector

AddOnly Mode

Post-install Configuration for AddOnly Mode: The AddOnly mode of the RAC Connector requires that the binaries of the Agent Controller (.dll, .so) are available to the dynamic library loader of the operating system. The preferred way to achieve this is to include the binaries folder of the Agent Controller in the PATH environment variable on Windows platforms, and in the LD_LIBRARY_PATH environment variable on Linux platforms. This can be done either globally or just for the process of the Tivoli Directory Integrator Server. For example:

- On Windows: modify the PATH environment variable from "My Computer" -> "Properties" -> "Advanced" -> "Environment variables"; add the required path to the Agent Controller libraries.

- On Linux: add lines like the following in the startup scripts (ibmdisrv and ibmditk) after the PATH definition and before the startup line:

  ```
  LD_LIBRARY_PATH=/AgentController/lib
  export LD_LIBRARY_PATH
  ```

  If you have LD_LIBRARY_PATH elements of your own, add these to the LD_LIBRARY_PATH definition.

When operating in AddOnly mode, the first RAC Connector on the Tivoli Directory Integrator Server registers a Logging Agent with the local Agent Controller.

All Common Base Event objects received from the AssemblyLine, are serialized as XML and written to the Logging Agent. The Logging Agent stays operational as long as the master process of the Tivoli Directory Integrator server is alive. During its lifetime it can be monitored by clients even if the Connector which registered it has already closed. When the Tivoli Directory Integrator server stops (or crashes), however, the Agent Controller (RAC) terminates the Tivoli Directory Integrator Logging Agent's registration.

The Connector will wait a specified amount of time for a monitoring client to arrive before starting to write data to the Logging Agent. In particular, it can wait forever. This is specified by the **Wait to be monitored** Connector parameter. When a client starts monitoring the agent, the agent starts transferring data to the Agent Controller. The Agent Controller then sends the data to the client.

Waiting happens before each Connector write attempt.

If the waiting time expires and there is still no monitoring client, the Connector throws an Exception. However, if a client starts monitoring the agent while the Connector is waiting, the waiting is interrupted and the agent starts transferring data to the Agent Controller.

Depending on the **Wait to be monitored** Connector parameter value the Connector could potentially wait indefinitely for a client to start monitoring the agent. This would cause the entire AssemblyLine to block indefinitely. Precisely for this reason the following Connector method is available to you:

```
public boolean isLogging();
```

This method returns *true* if there is a client monitoring/listening for data from this Connector and *false* otherwise. This method is accessible through JavaScript and can be invoked on the Connector object (that is, `thisConnector.islogging()`.)

You can use this method in order to detect whether the Connector will block when the AssemblyLine execution reaches the Connector. If blocking is not desirable, but loosing data is unacceptable, then you could implement a solution which temporarily stores the data into a queue (possibly the Tivoli Directory Integrator Memory Queue) when the isLogging() method returns false.

Iterator Mode

In Iterator mode the RAC Connector acts as a client of a remote Agent Controller. It connects to the Agent Controller to obtain a handle to the Logging Agent, whose name is specified in the Connector's configuration. After that the Connector starts monitoring the Logging Agent. During the monitoring, the Connector receives data produced by the Logging Agent.

Data reception is handled asynchronously by the Agent Controller client library and queued there. The Connector is notified when data reception occurs, and when the Connector reads from the queue a buffer is received with the incoming binary data. The queue is blocking, so the Connector will wait if no data is available and the data processor will wait if there is no free space in the queue.

The received binary data contains a CommonBaseEvent object serialized as XML in UTF-8 encoding. In addition, the CommonBaseEvent is decoded from the buffer and made available to the Connector in the Input Map.

If there is no active agent with the specified name when the Connector contacts the Agent Controller, the Connector waits until such an agent is registered.

If at some point the agent gets deregistered (while the Connector is listening for events), the Connector will wait for another agent with the same name to appear. Essentially the Connector never stops unless its connection to the Agent Controller fails.

The Connector exposes a method, which provides access to the Common Base Event object obtained by the Connector on the current AssemblyLine iteration (the last event, processed by the 'getNextEntry' method of the Connector):

```
public CommonBaseEvent getCurrentCbeObject();
```

Schema

The connector internally uses the “CBE Function Component” on page 395, and uses that particular FC's schema.

See also

Agent Controller: overview, architecture, administration and configuration,
TPTP Data Collection Framework. How to develop agents and clients using Java/C++,
Monitoring an application with logging agents,
Log and Trace Analyzer,
“GLA Connector” on page 103.

RDBMS Change Detection Connector

The RDBMS Change Detection Connector enables IBM Tivoli Directory Integrator to detect when changes have occurred in specific RDBMS tables. Currently, setup scenarios are provided for tables in Oracle, DB2, MS SQL, Informix and Sybase databases.

RDBMS's have no common mechanism to inform the outside world of the changes that have been taking place on any selected database table. To address this shortcoming, IBM Tivoli Directory Integrator assumes that some RDBMS mechanism (such as a trigger, stored procedures or other) is able to maintain a separate change table containing one record per modified record in the target table. Sequence numbers are also maintained by the same mechanism.

Similar to an LDAP Change Detection Connector, the RDBMS Change Detection Connector communicates with the change table that is structured in a specific format that enables the connector to propagate changes to other systems. The format is the same that IBM DB2 Information Integrator (version 8) uses, providing IBM Tivoli Directory Integrator users with the option to use DB2II to create such tables, or create the tables in some other manner. The RDBMS Change Detection Connector keeps track of a sequence number so that it only reports changes since the last iteration through the change table.

The RDBMS Change Detection Connector uses JDBC to connect to a specific RDBMS table. See the "JDBC Connector" on page 133 for more information about JDBC driver issues.

The RDBMS Change Detection Connector only operates in Iterator mode.

This connector supports Delta Tagging at the Entry level only.

The RDBMS Change Detection Connector reads specific fields to determine new changes in the change table (see "Change table format" on page 247). The RDBMS Change Detection Connector reads the next change table record, or discovers the first change table record. If the RDBMS Change Detection Connector finds no data in the change table, the RDBMS Change Detection Connector checks whether it has exceeded the maximum wait time. If the RDBMS Change Detection Connector has exceeded the maximum wait time, it returns **null** to signal end of the iteration. If the RDBMS Change Detection Connector finds no data in the change table, and has not exceeded the maximum wait time, it waits for a specific number of seconds (**Poll Interval**), then reads the next change table record.

If the Connector returns data in the change table, the RDBMS Change Detection Connector increments and updates the **nextchangelog** number in the User Property Store (an area in the System Store tailored for this type of persistent information).

For each Entry returned, control information (counters, operation, time/date) is moved into Entry properties. All non-control information fields in the change table are copied as is to the Entry as attributes. The Entry objects operation (as returned by **getOperation**) is set to the corresponding changelog operation (Add, Delete or Modify).

This Connector in principle can handle secure connections using the SSL protocol; but it may require driver specific configuration steps in order to set up the SSL support. Refer to manufacturer's driver documentation for details.

Configuration

The Connector needs the following parameters:

JDBC URL
: See documentation for your JDBC provider. This is the JDBC URL to the target database.

Username
This is the user ID with which the Connector signs on to the RDBMS. Only the tables available to this user are shown.

Password
The password for the user. It is used to authenticate to the RDBMS using the username/password authentication mechanism.

Schema
The schema (that is, the owner) of the table of the database that you want to monitor. If left blank, the value of the **Username** parameter is used.

JDBC Driver
The JDBC driver class name. The default value for this parameter is `com.ibm.db2.jcc.DB2Driver`.

Table Name
The table or view to monitor for changes.

Remove Processed Rows
Select to remove all previously processed table rows before the next poll attempt. This cleanup is done when Iterator State is persisted.

Iterator State Key
Specifies the name of the parameter that stores the current synchronization state in the User Property Store of the IBM Tivoli Directory Integrator. This must be a unique name for all parameters stored in one instance of the IBM Tivoli Directory Integrator User Property Store.

The **Delete** button will delete this state information from the User Property Store.

Start At
This parameter is only taken into consideration if the **Iterator State Key** is not found in the property store or is left blank. It indicates the position of the record in the "change table" from which the connector will start reading entries. The parameter accepts values between 1 and EOD (End Of Data - the number of the last record in the "change table"). If the provided input value for the parameter is not valid, an appropriate exception will be thrown at runtime.

State Key Persistence
Governs the method used for saving the Connector's state to the System Store. The default and recommended setting is **End of Cycle**, and choices are:

After Read
Updates the System Store when you read an entry from the RDBMS Server change log, before you continue with the rest of the AssemblyLine.

End of Cycle
Updates the System Store when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual
Switches off the automatic updating of the System Store with this Connector's state information; instead, you will need to save the state by manually calling the RDBMS Change Detection Connector's *saveStateKey()* method, somewhere in your AssemblyLine.

Sleep Interval
Specifies the time (in seconds) that IBM Tivoli Directory Integrator waits between polls of the change table.

Timeout
Specifies the time (in seconds) to wait for new changes. A value of **0** (zero) causes the Connector to wait indefinitely.

Commit
Controls when database transactions are committed. Options are:

- **After every database operation**
- **On Connector close**
- **Manual**

Manual means user must call `commit()`.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Change table format

This example change table captures the changes from a table containing the fields NAME and EMAIL. Elements in bold are common for all Changelog table. The syntax for this example is for Oracle.

```
IBMSNAP_COMMITSEQ is used as our changelog-nr.
IBMSNAP_OPERATION takes on of the values I (Insert), U (Updated)  or D (Deleted).
CREATE TABLE "SYSTEM"."CCDCHANGELOG"
(
IBMSNAP_COMMITSEQ   RAW(10)   NOT NULL,
IBMSNAP_INTENTSEQ   RAW(10)   NOT NULL,
IBMSNAP_OPERATION   CHAR(1)   NOT NULL,
IBMSNAP_LOGMARKER   DATE      NOT NULL,
NAME   VARCHAR2 ( 80 )  NOT NULL,
EMAIL   VARCHAR2 ( 80 )
)#
```

The RDBMS Change Detection Connector does not work if the **ibmsnap_commitseq** column name used internally in the connector does not match exactly with the actual column in the database. This is true only when case-sensitivity is turned on for data objects in the Database the RDBMS Change Detection Connector is iterating on.

To handle this the column name is externalized as a connector configuration parameter. This provides the DBA an easy way to set **ibmsnap_commitseq** with the same case as used in his Database table. However, this parameter is not visible in connector config tab. To configure this parameter, you will have to set this manually in the *before initialize* hooks of the RDBMS Change Detection Connector. This will enable multiple RDBMS Change Detection Connectors to have their own copy of the column name value set for the change table the connector iterates on. For example,

```
myConn.connector.setParam("rdbms.chlog.col","IBMSNAP_COMMITSEQ");
```

sets the name of the **ibmsnap_commitseq** column to literally, **IBMSNAP_COMMITSEQ**. The default is lowercase otherwise.

Creating change tables in DB2

The following example creates triggers in a DB2 database to maintain the change table as described previous:

```
connect to your_username

drop table email
drop table ccdemail

create table email ( \
 name varchar(80), \
 email varchar(80) \
)

create table ccdemail ( \
 ibmsnap_commitseq integer, \
 ibmsnap_intentseq integer, \
 ibmsnap_logmarker date, \
 ibmsnap_operation char, \
 name varchar(80), \
```

```
 email varchar(80) \
)

drop sequence ccdemail_seq
create sequence ccdemail_seq

create trigger t_email_ins after insert on email referencing new as n \
 for each row mode db2sql \
   INSERT INTO your_username.ccdemail VALUES (nextval for ccdemail_seq, 0,
   CURRENT_DATE, 'I', n.name, n.email )

create trigger t_email_del after delete on email referencing old as n \
 for each row mode db2sql \
   INSERT INTO your_username.ccdemail VALUES (nextval for ccdemail_seq, 0,
   CURRENT_DATE, 'D', n.name, n.email )

create trigger t_email_upd after update on email referencing new as n \
 for each row mode db2sql \
   INSERT INTO your_username.ccdemail VALUES (nextval for ccdemail_seq, 0,
   CURRENT_DATE, 'U', n.name, n.email )
```

Creating change tables in Oracle

Given that your username is "ORAID", then this (example) change table will capture the changes from a table containing the fields NAME and EMAIL. Boldfaced elements are common for all change tables. Bold faced entries are extra control information that will end up as Entry properties.

```
-- create source email table in Oracle.
---This will be the table that the RDBMS Change Detection Connector will detect changes on.
CREATE TABLE ORAID.EMAIL
(
 NAME VARCHAR2(80),
 EMAIL VARCHAR2(80)
);
-- Sequence generators used for Intentseq and commitseq
CREATE SEQUENCE ORAID.SGENERATOR001
MINVALUE 100 INCREMENT BY 1 ORDER;

CREATE SEQUENCE ORAID.SGENERATOR002
MINVALUE 100 INCREMENT BY 1 ORDER;

-- create change table and index for email table
CREATE TABLE ORAID.CCDEMAIL
(
 IBMSNAP_COMMITSEQ   RAW(10)   NULL,
 IBMSNAP_INTENTSEQ   RAW(10)   NOT NULL,
 IBMSNAP_OPERATION   CHAR(1)   NOT NULL,
 IBMSNAP_LOGMARKER   DATE      NOT NULL,
 NAME VARCHAR2( 80 ),
EMAIL VARCHAR2( 80 )
);

CREATE UNIQUE INDEX ORAID.IXCCDEMAIL ON ORAID.CCDEMAIL
(
IBMSNAP_INTENTSEQ
);

-- create TRIGGER to capture INSERTs into email
CREATE TRIGGER  ORAID.EMAIL_INS_TRIG
AFTER INSERT ON ORAID.EMAIL
FOR EACH ROW BEGIN INSERT INTO ORAID.CCDEMAIL
( NAME,
  EMAIL,
  IBMSNAP_COMMITSEQ,
  IBMSNAP_INTENTSEQ,
  IBMSNAP_OPERATION,
  IBMSNAP_LOGMARKER )
```

```
 VALUES (
  :NEW.NAME,
  :NEW.EMAIL,
  LPAD(TO_CHAR(ORAID.SGENERATOR001.NEXTVAL),20,'0'),
  LPAD(TO_CHAR(ORAID.SGENERATOR002.NEXTVAL),20,'0'),
  'I',
  SYSDATE);END;


-- create TRIGGER to capture DELETE ops on email
CREATE TRIGGER  ORAID.EMAIL_DEL_TRIG
AFTER DELETE ON ORAID.EMAIL
FOR EACH ROW BEGIN INSERT INTO ORAID.CCDEMAIL
( NAME,
  EMAIL,
  IBMSNAP_COMMITSEQ,
  IBMSNAP_INTENTSEQ,
  IBMSNAP_OPERATION,
  IBMSNAP_LOGMARKER)
 VALUES
( :OLD.NAME,
  :OLD.EMAIL,
  LPAD(TO_CHAR(ORAID.SGENERATOR001.NEXTVAL),20,'0'),
  LPAD(TO_CHAR(ORAID.SGENERATOR002.NEXTVAL),20,'0'),
  'D',
  SYSDATE);END;


-- create TRIGGER to capture UPDATEs on email
CREATE TRIGGER  ORAID.EMAIL_UPD_TRIG
AFTER UPDATE ON ORAID.EMAIL
FOR EACH ROW BEGIN INSERT INTO ORAID.CCDEMAIL
( NAME,
  EMAIL,
  IBMSNAP_COMMITSEQ,
  IBMSNAP_INTENTSEQ,
  IBMSNAP_OPERATION,
  IBMSNAP_LOGMARKER )
 VALUES (
  :NEW.NAME,
  :NEW.EMAIL,
  LPAD(TO_CHAR(ORAID.SGENERATOR001.NEXTVAL),20,'0'),
  LPAD(TO_CHAR(ORAID.SGENERATOR002.NEXTVAL),20,'0'),
  'U',
  SYSDATE);END;
```

Creating change table and triggers in MS SQL

```
-- Source table msid.email.
-- This will be the table that the RDBMS Change Detection Connector will detect changes on.
CREATE TABLE msid.email
(
 NAME   VARCHAR (80),
 EMAIL   VARCHAR (80)
 );

-- CCD table to capture changes. The RDBMS Change Detection Connector uses the CCD table to capture
-- all the changes in the source table. This table needs to be created in the following format.
CREATE TABLE msid.ccdemail
(
 IBMSNAP_MSTMSTMP timestamp,
 IBMSNAP_COMMITSEQ   BINARY(10)   NOT NULL,
 IBMSNAP_INTENTSEQ   BINARY(10)   NOT NULL,
 IBMSNAP_OPERATION   CHAR(1)      NOT NULL,
```

```
 IBMSNAP_LOGMARKER   DATETIME     NOT NULL,
 NAME   VARCHAR (80),
 EMAIL   VARCHAR (80)
);
```

You also need to create triggers to capture the insert, update and delete operations performed on the email table.

```
CREATE TRIGGER  msid.email_ins_trig ON msid.email
FOR INSERT AS
BEGIN
 INSERT INTO msid.ccdemail
(NAME,
 EMAIL,
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER )
 SELECT
 NAME,
 EMAIL,
  @@DBTS,
 @@DBTS,
 'I',
 GETDATE() FROM inserted
END;
```

Note: : @@DBTS returns the value of the current timestamp data type for the current database. This timestamp is guaranteed to be unique in the database.

```
-- creating DELETE trigger to capture delete operations on email table
CREATE TRIGGER  msid.email_del_trig ON msid.email
FOR DELETE AS
BEGIN
 INSERT INTO msid.ccdemail
(
 NAME,
 EMAIL,
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER
)
 SELECT
 NAME,
 EMAIL,
 @@DBTS,
 @@DBTS,
 'D',
 GETDATE() FROM deleted
END;#

-- creating UPDATE trigger to capture update operations on email table
CREATE TRIGGER  msid.email_upd_trig ON msid.email
FOR UPDATE AS
BEGIN
 INSERT INTO msid.ccdemail
(
 NAME,
 EMAIL,
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER
)
 SELECT
 NAME,
```

```
 EMAIL,
 @@DBTS,
 @@DBTS,
 'U',
 GETDATE() FROM updated
END;
```

Creating change table and triggers in Informix

```
-- Create Source table infxid.email. This will be the table that the RDBMS Change Detection Connector
-- will detect changes on.
CREATE TABLE infxid.email
(
NAME VARCHAR(80),
EMAIL VARCHAR(80)
);

-- create ccdemail table to capture DML operations on email table
CREATE TABLE infxid.ccdemail
(
IBMSNAP_COMMITSEQ   CHAR(10)   NOT NULL,
IBMSNAP_INTENTSEQ   CHAR(10) NOT NULL,
IBMSNAP_OPERATION   CHAR(1)   NOT NULL,
IBMSNAP_LOGMARKER   DATETIME YEAR TO FRACTION(5) NOT NULL,
NAME   VARCHAR(80),
EMAIL   VARCHAR(80)
);

--Create sequence generators
CREATE SEQUENCE infxid.SG1
MINVALUE 100 INCREMENT BY 1;
CREATE SEQUENCE infxid.SG2
MINVALUE 100 INCREMENT BY 1;

-- procedure to capture INSERTs into email table
CREATE PROCEDURE infxid.email_ins_proc
(
NNAME  VARCHAR(80),

NEMAIL  VARCHAR(80)
)

DEFINE VARHEX CHAR(256);

 INSERT INTO infxid.ccdemail
(NAME,
 EMAIL,
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER )
 VALUES
(NNAME,
 NEMAIL,
 infxid.SG1.NEXTVAL,
 infxid.SG2.NEXTVAL,
 'I',
 CURRENT YEAR TO FRACTION(5));END PROCEDURE;

-- now create the trigger for INSERTs into ccdemail
CREATE TRIGGER  infxid.email_ins_trig
 INSERT ON infxid.email
 REFERENCING NEW AS NEW FOR EACH ROW( EXECUTE PROCEDURE
 infxid.email_ins_proc
( NEW.NAME,
  NEW.EMAIL
) );
```

```
-- create procedure to capture DELETEs on email table
CREATE PROCEDURE infxid.email_del_proc
(
 ONAME  VARCHAR(80),
 OEMAIL  VARCHAR(80)
 );

 INSERT INTO infxid.ccdemail
(NAME,
 EMAIL,
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER )
 VALUES
(ONAME,
 OEMAIL,
 infxid.SG1.NEXTVAL,
 infxid.SG2.NEXTVAL,
 'D',
 CURRENT YEAR TO FRACTION(5));END PROCEDURE;

-- create DELETE trigger
CREATE TRIGGER  infxid.email_del_trig
 DELETE ON infxid.email
 REFERENCING OLD AS OLD FOR EACH ROW( EXECUTE PROCEDURE
 infxid.email_del_proc
(OLD.NAME,
 OLD.EMAIL
) );

-- create PROCEDURE to capture updates
CREATE PROCEDURE infxid.email_upd_proc
(
 NNAME  VARCHAR(80),
 NEMAIL  VARCHAR(80)
);
 INSERT INTO infxid.ccdemail
(NAME,
 EMAIL,
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER)
 VALUES
(NNAME,
 NEMAIL,
 infxid.SG1.NEXTVAL,
 infxid.SG2.NEXTVAL,
 'U',
 CURRENT YEAR TO FRACTION(5));END PROCEDURE;

-- create TRIGGER to capture UPDATES
CREATE TRIGGER  infxid.email_upd_trig
 UPDATE ON infxid.email
 REFERENCING NEW AS NEW OLD AS OLD FOR EACH ROW( EXECUTE PROCEDURE
 infxid.email_upd_proc
(NEW.NAME,
 NEW.EMAIL
 ) );
```

Creating change table and triggers for SYBASE

```
-- Create Source table sybid.email.
-- This will be the table that the RDBMS Change Detection Connector will detect changes on.
CREATE TABLE sybid.EMAIL
```

```
(
 NAME   VARCHAR (80),
 EMAIL   VARCHAR (80)
)

-- Create CCD table to captures changes on email table
CREATE TABLE sybid.CCDEMAIL
(
 IBMSNAP_TMSTMP TIMESTAMP,
 IBMSNAP_COMMITSEQ  NUMERIC(10)   IDENTITY,
 IBMSNAP_INTENTSEQ   BINARY(10)   NOT NULL,
 IBMSNAP_OPERATION   CHAR(1)      NOT NULL,
 IBMSNAP_LOGMARKER   DATETIME     NOT NULL,
 NAME   VARCHAR(80),
 EMAIL   VARCHAR(80)
)

-- Create TRIGGER to capture INSERTs on email table
CREATE TRIGGER  sybid.EMAIL_INS_TRIG ON sybid.EMAIL
FOR INSERT AS
BEGIN
 INSERT INTO sybid.CCDEMAIL
(NAME,
 EMAIL,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER )
 SELECT
 NAME,
 EMAIL,
@@DBTS,
 'I',
 GETDATE() FROM inserted
END

NOTE: @@DBTS is a special database variable that yields the next database timestamp value

-- create TRIGGER to captures DELETE ops on EMAIL table
CREATE TRIGGER  sybid.EMAIL_DEL_TRIG ON sybid.EMAIL
FOR DELETE AS
BEGIN
 INSERT INTO sybid.CCDEMAIL
(
 NAME,
 EMAIL,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER
)
 SELECT
 NAME,
 EMAIL,
 @@DBTS,
 'D',
 GETDATE() FROM deleted
END

-- create TRIGGER to capture UPDATEs on email
CREATE TRIGGER  sybid.EMAIL_UPD_TRIG ON sybid.EMAIL
FOR UPDATE AS
BEGIN
  DECLARE @COUNTER INT
  SELECT @COUNTER=COUNT(*) FROM deleted
  IF @COUNTER>1
  BEGIN
  DECLARE @NAME  VARCHAR ( 80 )
  DECLARE @EMAIL  VARCHAR ( 80 )
```

```
 DECLARE insertedrows CURSOR FOR SELECT * FROM inserted
 OPEN insertedrows
 WHILE 1=1 BEGIN
 FETCH insertedrows INTO
 @NAME,
 @EMAIL
  IF @@fetch_status<>0 BREAK
 ELSE INSERT INTO sybid.CCDEMAIL
(
 NAME,
 EMAIL,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER
)
VALUES
(
 @NAME,
 @EMAIL,
 @@DBTS,
'U',
GETDATE()
)
END
 DEALLOCATE insertedrows
END ELSE INSERT INTO sybid.CCDEMAIL(
 NAME,
 EMAIL,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER
)
SELECT
 I.NAME,
 I.EMAIL,
 @@DBTS,
'U',
GETDATE() FROM inserted I
END
```

Example

An example is provided under the directory *TDI_install_dir*/examples/RDBMS. The example demonstrates the abilities of the RDBMS Change Detection Connector to detect changes over a table in a remote DataBase. The current example is designed to work with IBM DB2 only.

Script Connector

The Script Connector enables you to write your own Connector in JavaScript.

A Script Connector must implement a few functions to operate. If you plan to use it for iteration purposes only (for example, reading, not searching or updating), you can operate with two functions only. If you plan to use it as a fully qualified Connector, you must implement all functions. The functions do not use parameters. Passing data between the hosting Connector and the script is enabled by using predefined objects. One of these predefined objects is the **result** object, which is used to communicate status information. Upon entry in either function, the **status** field is set to **normal**, which causes the hosting Connector to continue calls. Signaling **end-of-input** or **error** is done by setting the **status** and **message** fields in this object. Two other script objects are defined upon function entry, the **entry** object and the **search** object.

Note: When you modify a Script Connector or Parser, the script gets copied from the Library where it is stored, into your configuration file. This enables you to customize the script, but with the caveat that new versions are not known to your AssemblyLine.

One workaround is to remove the old Script Connector from the AssemblyLine and reintroduce it.

Predefined script objects

main

The Config Instance (RS object) that is running.

task

The AssemblyLine this Connector is a part of

system

A UserFunctions object.

The result object

setStatus (code)

- **0** - End of Input
- **1** - Status OK
- **2** - Error

setMessage (text)

Error message.

The config object

This object gives you access to the configuration of this AL component, and its Input and Output schema — note that the getSchema() method of this object has a single Boolean parameter: *true* means to return the Input Schema while false gets *you* the Output Schema.

The entry object

The **entry** object corresponds to the **conn** Entry for a Connector (or Function, when scripting an FC.)

See "The Entry object" on page 524 for more details.

The search object

The **search** object gives you access to the searchCriteria object (built based on Link Criteria settings.) See "The Search (criteria) object" on page 527 for more details.

The connector object

A reference to this Connector.

This could be useful for example when returning multiple Entries found in the findEntry() function, with code similar to this:

```
function findEntry() {
 connector.clearFindEntries();
 // Use the search object fo find Entries, and
 for ( entry = all Entries found) {
  connector.addFindEntry(entry)
 }
 if (connector.getFindEntryCount() == 1)
  result.setStatus(1);
 else
  result.setStatus(0);
}
```

Functions

The following functions can be implemented by the Script Connector. Even though some functions might never be called, it is recommended that you insert the functions with an error-signaling code that notifies the caller that the function is unsupported.

initialize
: This function initializes the Connector. It is called before any of the other functions and should contain code that initializes basic parameters, establishes connections, and so forth.

selectEntries
: This function is called to prepare the Connector for sequential read. When this function is called it is typically because the Connector is used as an Iterator in an AssemblyLine.

getNextEntry
: This function must populate the Entry object with attributes and values from the next entry in the input set. When the Connector has no more entries to return, it must use the **result** object to signal end-of-input back to the caller.

findEntry
: The **findEntry** function is called to find an entry in the connected system that matches the criteria specified in the **search** object. If the Connector finds a single matching entry, then the Connector populates the **entry** object. If no entries are found, the Connector must set the error code in the **result** object to signal a failure to find the entry. If more than one entry is found, then the Connector might populate the array of duplicate entries. Otherwise, the same procedure is followed as when there are no entries found.

modEntry
: This function is called to modify an existing entry in the connected system. The new entry data is given by the **entry** object, and the **search** object specifies which entry to modify. Some Connectors might silently ignore the **search** object, and use the **entry** object to determine which entry to modify.

putEntry
: This function adds the **entry** object to the connected system.

deleteEntry
: This function is called to delete an existing entry in the connected system. The **search** object specifies which entry to delete. Some Connectors might silently ignore the **search** object, and use the **entry** object to determine which entry to delete.

queryReply
: This function is called when the Connector is used in Call/Reply mode.

querySchema
: The querySchema() function is used to discover schema for this Connector. If implemented, a

vector of **Entry** objects is returned for each column/attribute it discovered. The querySchema() function is only called when you "Open/Query" in the attribute map (not when you click the quick discovery button).

In order to support Schema discovery your Script Connector or -FC can contain code like this:

```
function querySchema() {
  config.getSchema(true).newItem("name-in");
  config.getSchema(true).newItem("address-in");
  config.getSchema(false).newItem("name-out");
  config.getSchema(false).newItem("address-out");
 }
```

This would create two items in the input and output schemas respectively. Check the SchemaConfig and SchemaItemConfig API (in the Javadocs) for more details.

terminate
This function is called when the Connector has finished its task. It should contain code that frees up any resources (for example, locks, connections) taken during its work.

According to the various modes, these are the minimum required functions you need to implement:

Table 19. Required functions

Mode	Function you must implement
Iterator	selectEntries() getNextEntry()
AddOnly	putEntry()
Lookup	findEntry()
Delete	findEntry() deleteEntry()
Update	findEntry() putEntry() modEntry()
CallReply	queryReply()

Configuration

The Connector needs the following parameters:

Edit Script...
This button opens a window where you can create your own script code. An empty skeleton will already be present.

Keep Global State
If this parameter is checked (default), then any global variables defined in the script will be kept after the Connector terminates, and they will be present with their last values when the Connector is re-initialized.

Note: With this parameter checked (which is the default) the behavior of this Connector changed in Tivoli Directory Integrator v7.1. If you need global variables to be reinitialized along with the Connector, you should either uncheck this parameter, or set these variables inside the `initialize()` method.

External Files
If you want to include external script files at runtime, specify them here. Specify one file on each line. These files are started before your script.

Include Global scripts
Include global scripts from the Script Library.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Examples

Navigate to the *root_directory*/examples/script_connector directory of your IBM Tivoli Directory Integrator installation.

See also

“Script Parser” on page 331,
“Scripted Function Component” on page 393,
"JavaScript Connector" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.

SNMP Connector

This Connector listens for SNMP traps sent on the network and returns an entry with the name and values for all elements in an SNMP PDU.

Notes:

1. In Client mode, a request is retried 5 times with increasing intervals; the retry waiting period doubles on every retry, starting with 5 seconds. Timeout occurs if no answer is received.
2. If you want to send SNMP Traps, the `system.snmpTrap()` method is available.
3. The SNMP Connector does not support the Advanced Link Criteria (see "Advanced link criteria" in *IBM Tivoli Directory Integrator V7.1 Users Guide*).

Configuration

The Connector needs the following parameters:

Community String
Use **public** to test the Connector.

Mode Trap Listener or Client. The Client mode can use Connectors in AddOnly mode (SNMP Set), Lookup mode (SNMP Get) or Iterator mode (Walk).

Trap listener can only Iterate, listening to traps on the local host.

SNMP Trap Port
Port in Trap mode. Unused in Client mode.

Trap wait timeout
Timeout in Trap mode. The number of milliseconds to wait for the next Protocol Data Unit (PDU). If value is zero or less, the Connector waits forever.

SNMP Host (for get)
Only used for Get in Client mode. Not used in Trap mode.

SNMP Port
Client port (for client mode). Not used in Trap mode.

SNMP Walk OID (iterate)
Used only in Client mode, Iterator Connector. Indicates the OID tree to walk.

SNMP Version
The default version for get/walk is the Client mode. Unused in trap mode.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Note: Link Criteria are treated differently for this Connector. In Lookup mode, the connector performs a get request returning the **oid/value** for the requested oid. The link criteria specifies **oid**, as well as **server**, **port** and **version**. An example link criterion might be **"oid" = "1.1.1.1.1.1.1"**.

Examples

Go to the *root_directory*`/examples/snmpTrap` directory of your IBM Tivoli Directory Integrator installation.

See also

SNMP V1: RFC1155, RFC1157
SNMP V2: RFC1901, RFC1907, RFC2578
SNMP V3: RFC3411, RFC3412.

SNMP Server Connector

The SNMP Server Connector supports SNMP v1. SNMP v2 is supported without the SNMP v2 authentication and encryption features.

The Connector does not support SNMP TRAP messages.

The SNMP Server Connector operates in server mode only. The transport protocol it uses is UDP and not TCP. UDP is an unreliable transport protocol, and SSL cannot run on top of an unreliable transport protocol. That is why the Connector cannot use SSL to protect the transport layer.

The SNMP Server Connector (contrary to other Connectors in Server Mode) uses DatagramSockets. That is why there is no notion of connection. The SNMP Server Connector uses a single DatagramSocket which receives SNMP packets from many different SNMP managers on the network.

In the getNextClient() method, the socket blocks on the receive() method until an SNMP packet is received. Then, the Connector creates a new instance of itself, passes the received packet to the child Connector and returns the child Connector.

The getNextEntry() method extracts the SNMP request packet attributes and sets them in the *conn* Entry, ready for Input Attribute Mapping.

The replyEntry() method extracts the Attributes from the *conn* Entry and creates an SNMP response packet and returns it to the client; the *conn* Entry should be populated using Output Attribute Mapping.

The replyEntry() method uses the parent Connector's DatagramSocket to send back the response. Since the parent Connector's DatagramSocket is shared among all child Connectors the access to the DatagramSocket is synchronized.

Connector Schema

The SNMP Server Connector makes the following Attributes available for Input Attribute Mapping:

snmp.operation
: java.lang.String object, which represents the SNMP operation invoked. The supported operation types are GET, GETNEXT and SET.

snmp.community
: Defines an access environment for a group of Network Management Systems (NMSs). NMSs within the community are said to exist within the same administrative domain. Community names serve as a weak form of authentication because devices that do not know the proper community name are precluded from SNMP operations.

snmp.remoteip
: IP address of the SNMP client (dot notation).

snmp.errorcode
: Indicates one of a number of errors and error types. Only the response operation sets this field. Other operations set this field to zero.

snmp.errorindex
: Associates an error with a particular object instance. Only the response operation sets this field. Other operations set this field to zero.

snmp.request-id
: Associates SNMP requests with responses.

snmp.PDU
: Protocol Data Unit. SNMP PDUs contain a specific command (Get, Set, etc.) and operands that indicate the object instances involved in the transaction.

snmp.oid
OID is an address of a MIB structure, indicating a specific variable or attribute to be read or modified in the target system. A GET can contain a list of OIDs, while a SET can also include the corresponding values to be set for those variables in the target system. However, most SNMP deployments use only one OID per SNMP message.

snmp.oidvalue
Contains the corresponding value of one OID. This is a String representation.

snmp.oidvalue.raw
Contains the corresponding value of one OID. This is an Object representation.

Configuration

The SNMP Server Connector uses the following parameters:

UDP Port
Specifies the UDP port on which the Connector (1) receives incoming SNMP request packets and from which (2) sends SNMP response packets. The default value is 161, which is the standard port for SNMP GET/SET operations.

Verify Community
Specifies the SNMP Community name. SNMP Community names serve as a weak form of authentication because devices that do not know the proper community name are precluded from SNMP operations.

If set, the Connector discards all messages not matching this community string. If blank, the Connector allows all community strings.

The default value is "public".

Detailed Log
If enabled, will generate detailed Log messages.

Tivoli Access Manager (TAM) Connector

Introduction

The IBM Tivoli Directory Integrator 7.1 Connector for Tivoli Access Manager enables the provisioning and management of Tivoli Access Manager User accounts, Groups, Policies, Domains, SSO Resources, SSO Resource Groups, and SSO User Credentials to external applications (with respect to Tivoli Access Manager). The Connector uses the Tivoli Access Manager Java API.

The key features and benefits of the Connector are:

- Support for Create, Read, Update, and Delete operations for Tivoli Access Manager User accounts, Groups, Policies, Domains, SSO Resources, SSO Resource Groups, and SSO User Credentials.

Note: The Connector uses the TAM 6 Java API to manipulate the attributes of the targeted TAM objects. Therefore, this Connector can't support TAM 5.1 because of JRE support restrictions for the TAM 5.1 Runtime Environment (RTE). It supports TAM 6.0 and TAM 6.1 only.

SSL communication with the TAM Server is supported.

Connector Modes

The Connector supports the Lookup, Iterator, Update, AddOnly, and Delete modes. Refer to "Using the Connector" on page 266 for specific usage of the various modes.

Skip Lookup in Update and Delete mode

The TAM Connector supports the **Skip Lookup** general option in Update or Delete mode. When it is selected, no search is performed prior to actual update and delete operations.

Valid Link Criteria must be present, that is, the mandatory attribute must be defined in the Link Criteria of the Connector, as defined in the tables of mandatory attributes under the "Update Mode" on page 269 and "Delete Mode" on page 271 sections respectively.

Configuration

Before attempting to use the connector in an AssemblyLine, Tivoli Access Manager version 6.x must be installed on the target machine: The Tivoli Access Manager Java Runtime Environment (JRTE) must also be installed on the same machine as Tivoli Directory Integrator.

Configuring the Tivoli Access Manager Java Run Time

The Connector makes use of the Tivoli Access Manager Java API and therefore the Tivoli Access Manager Runtime for Java must be installed on the Tivoli Directory Integrator machine. For information on how to install and configure Tivoli Access Manager Runtime for Java on the Tivoli Directory Integrator machine, refer to the *Tivoli Access Manager Installation Guide*.

When entering the parameters to the configuration utility (**pdjrtecfg**):

- Check that both the policy server and registry server are running.
- Ensure that Tivoli Directory Integrator is not running.
- Make sure that Tivoli Directory Integrator's JVM is in your path, for example with the following command (on UNIX/Linux):

  ```
  export PATH=/opt/IBM/TDI/V7.1/jvm/jre/bin:$PATH
  ```

- Specify the location of the Tivoli Directory Integrator JRE directory. For example, on a Linux machine the default Tivoli Directory Integrator JVM directory is:

  ```
  /opt/IBM/TDI/V7.1/jvm/jre
  ```

- Specify a configuration type of Full. For example, from the Policy_Director/sbin directory, enter the following command (as one line):

```
pdjrtecfg -action config -host TAM_host_name -port 7135
  -java_home "/opt/IBM/TDI/V7.1/jvm/jre" -config_type full
```

where *TAM_hostname* is the name of the host where Tivoli Access Manager Policy Server is installed. You should get the message "*Configuration of Access Manager Runtime for Java is in progress*". This might take several minutes. After completion, you should get the message "*Configuration of Access Manager Runtime for Java completed successfully*".

Configuring secure communication to the Tivoli Access Manager policy server

To configure secure communication between Tivoli Directory Integrator and Tivoli Access Manager policy server and authorization server, and for Tivoli Directory Integrator to become an authorized Tivoli Access Manager Java application, run the **SvrSslCfg** utility on the Tivoli Directory Integrator machine.

For example, from the *TDI_install_dir*/jvm/jre/bin directory, enter the following command (as one line). This command must be run with the Tivoli Directory Integrator's Java executable:

```
/opt/IBM/TDI/V7.1/jvm/jre/bin/java com.tivoli.pd.jcfg.SvrSslCfg -action config -admin_id sec_master
  -admin_pwd password -appsvr_id appsvr -host TAM_host_name -mode remote -port 999
  -policysvr policy_svr:7135:1 -authzsvr auth_svr:7136:1 -cfg_file cfg_file_name
  -key_file keyfile_name -cfg_action create
```

For complete information on the **SvrSslCfg** utility, refer to the *Tivoli Access Manager Authorization Java Classes Developer Reference* (specifically *Appendix A*).

Configuring SSL

The following steps allow you to optionally create a new self-signed certificate, and configure Tivoli Directory Integrator to use the certificate:

1. Open the IBM Tivoli Directory Integrator Configuration Editor.
2. Select **KeyManager** from the Toolbar. The IBM Key Manager tool opens.
3. Select **Key Database File** then **New**.
4. Select "JKS" as the Key database type.
5. Enter an appropriate **File Name** and an appropriate **Location**. Click **OK**.
6. Enter a **Password**. Enter the password again to confirm. Click **OK**.
7. In the Key database content section, select **Personal Certificates**. Click **New Self-Signed**.

   Note: Alternatively, an existing certificate can be used. If you wish to do this, click **Export/Import** to import the appropriate certificate.
8. Enter an appropriate **Key Label**, an appropriate **Organization**, and any other appropriate information. Click **OK**.
9. Close IBM Key Manager.
10. In the Tivoli Directory Integrator Configuration Editor, select **Browse Server Stores** then click on **Open** for the Server Store you wish to configure, usually Default.tdiserver. Double-click **Solution-Properties**, and the solution properties table is opened.
11. Locate the `javax.net.ssl.trustStore` parameter. Enter the value of Key database File created in step 5 above.
12. Locate the `javax.net.ssl.trustStorePassword` parameter. Enter the value of the Password entered in step 6 above.
13. Locate the `javax.net.ssl.trustStoreType` parameter. Enter "jks".
14. Locate the `javax.net.ssl.keyStore` parameter. Enter the value of Key database File created in step 5 above.
15. Locate the `javax.net.ssl.keyStorePassword` parameter. Enter the value of the Password entered in step 6 above.
16. Locate the `javax.net.ssl.keyStoreType` parameter. Enter "jks".

17. Click **Close** to close the solution properties table. The changes to the solution properties are saved in the relevant `solution.properties` file.
18. Close Tivoli Directory Integrator Configuration Editor.

Refer to *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide* for more information on configuring SSL.

Configuring the Connector

The Tivoli Directory Integrator Connector for Tivoli Access Manager can be added directly into an assembly line. The following section lists the configuration parameters that are available.

TAM ID
The Connector attempts to log on to Tivoli Access Manager with this user name and the password specified by the Password parameter. Default value: *sec_master*

TAM Password
The value of this parameter is taken in account only when the parameter **TAM ID** is set to a non-blank value. It then specifies the password used for the logon operation. The default value is blank.

Domain
Specifies the TAM Domain. The default is "Default". Pressing the "Domains" button next to this parameter queries the TAM Server for a list of Domains, from which you can select the appropriate one. The Connector attempts to log on to Tivoli Access Manager with the **TAM ID** and **TAM Password** parameters.

TAM Program Name
The name used by Tivoli Access Manager to identify this Connector. Default value: *IDI*

TAM Configuration File
File pathname for the Tivoli Access Manager configuration file created by the **SvrSslCfg** configuration utility.

Entry Type
Must be set to one of the following values:

- *User* (specifying that the Connector operates with data structured by Tivoli Access Manager Users),
- *Group* (specifying that the Connector operates with data structured by Tivoli Access Manager Groups),
- *Policy* (specifying that the Connector operates with data structured by Tivoli Access Manager User Policies),
- *Domain* (specifying that the Connector operates with data structured by Tivoli Access Manager Domains),
- *SSOCred* (specifying that the Connector operates with data structured by Tivoli Access Manager SSO Credentials),
- *SSOResource* (specifying that the Connector operates with data structured by Tivoli Access Manager SSO Resources),
- *SSResourceGroup* (specifying that the Connector operates with data structured by Tivoli Access Manager SSO Resource Groups).

Filter users/groups
An optional connector attribute that defines a filter string used to select "User" or "Group" object types. The parameter is only used in Iterator mode with one of those two Entry Types. By default, this attribute is empty, which implies no filtering.

Import Users/Groups from Registry
When checked, Tivoli Access Manager will import users/groups from the User Registry instead

of creating users in the User Registry during an **add** operation. In Update mode, users/groups will be imported through the **add** operation only, and not through the **modify** operation.

Delete Users/Groups/Domains from Registry
When checked, Tivoli Access Manager will delete users/groups/domains from the User Registry during a delete operation. The UserName, RegistryUID, Firstname and Lastname attributes are mandatory for this operation to find the correct LDAP registry user name of the TAM account to import.

In Update mode, users/groups will be imported through the add operation only, and not through the modify operation.

Detailed Log
If this field is checked, additional debug log messages are generated.

Using the Connector

This section describes how to use the Connector in each of the supported IBM Tivoli Directory Integrator Connector modes. The section also describes the Tivoli Directory Integrator Entry schema supported by the Connector.

Note: When the Connector executes in the Assembly line, a Tivoli Access Manager Context is created in the *Initialize* method of the Connector. For performance reasons, so that a Context is not created for every Tivoli Access Manager Connector Instance, the Tivoli Access Manager Connector should be cached (pooled) within the AssemblyLine. The caching of a Connector within the AssemblyLine can be configured within Tivoli Directory Integrator. Please refer to the *IBM Tivoli Directory Integrator V7.1 Users Guide* for more information.

When the Connector is configured to manipulate TAM Policy objects, special consideration is required when supply attribute values in the work entry that will feed the Connector in **AddOnly** or **Update** Modes. The policy object attributes are grouped together for related policy items. The attributes can be broken up into sets where each set of attributes requires a value to update or apply any of the individual attributes for that policy item. For example, when manipuilating the Policy item Account Expiry Date, you must supply values for each of the attributes AcctExpDateEnforced, AcctExpDateUnlimited, and AcctExpDate. If you wish to then modify any of these attributes for Account Expiry Date, you must again also supply values for each of the three attributes and the UserName attribute.

The following table defines the Policy items and their attribute groupings.

Table 20. Policy Items

Policy item	Set of Required Policy Entry Attributes
Account Expiry Date	AcctExpDateEnforced, AcctExpDateUnlimited, AcctExpDate.
Account Disable Time	AcctDisableTimeEnforced, AcctDisableTimeUnlimited, AcctDisableTime
Account Password Spaces	PwdSpacesAllowedEnforced, PwdSpacesAllowed
Account Maximum Password Age	MaxPwdAgeEnforced, MaxPwdAge
Account Maximum Repeat Characters	MaxPwdRepCharsEnforced, MaxPwdRepChars
Account Minimum Alphabetic Characters	MinPwdAlphasEnforced, MinPwdAlphas
Account Minimum Non-Alphabetic Characters	MinPwdNonAlphasEnforced, MinPwdNonAlphas
Account Time Of Day Access	TodAccessEnforced, AccessibleDays, AccessStartTime, AccessEndTime, AccessTimezone
Account Minimum Password Length	MinPwdLenEnforced, MinPwdLen
Account Maximum Failed Login Attempts	MaxFailedLoginsEnforced, MaxFailedLogins

Table 20. Policy Items (continued)

Policy item	Set of Required Policy Entry Attributes
Account Maximum Concurrent Web Sessions	MaxConcWebSessionsEnforced, MaxConcWebSessions, MaxConcWebSessionsUnlimited, MaxConcWebSessionsDisplaced

AddOnly Mode

When deployed in **AddOnly** mode, the Connector is able to create a range of data in the Tivoli Access Manager database. The Connector should be added to the **Flow** section of a Tivoli Directory Integrator AssemblyLine. The **Output Map** must define a mapping for the following attributes, these attributes can be also be retrieved through querying the Connector Schema.

Notes:

1. Attributes marked with an asterisk (*) are mandatory.
2. For a detailed description of all attributes, please refer to "Connector Input Attribute Details" on page 273.
3. Keep in mind the caveats on manipulating Policy items and their required Policy Entry attributes as stipulated in Table 20 on page 266.

Table 21. Attributes by Entry Type in AddOnly Mode

Entry Type	Attribute
User	UserName*
	RegistryUID*
	FirstName*
	LastName*
	Description
	Password*
	IsAccountValid
	IsPasswordValid
	IsSSOUser
	NoPasswordPolicyOnCreate
	MaxFailedLogins
	MaxConcWebSessions
	Groups (Multivalued attribute) - the User must not already be a member of the Group
Group	GroupName*
	RegistryGID*
	CommonName
	Description
	ObjectContainer
	Users (Multivalued attribute) - the Group must not already contain the User
Policy	UserName*
	AcctExpDateEnforced
	AcctExpDateUnlimited
	AcctExpDate

Table 21. Attributes by Entry Type in AddOnly Mode (continued)

Entry Type	Attribute
	AcctDisableTimeEnforced
	AcctDisableTimeUnlimited
	AcctDisableTimeInterval
	PwdSpacesAllowedEnforced
	PwdSpacesAllowed
	MaxPwdAgeEnforced
	MaxPwdAge
	MaxPwdRepCharsEnforced
	MaxPwdRepChars
	MinPwdAlphas
	MinPwdNonAlphasEnforced
	MinPwdNonAlphas
	TodAccessEnforced
	AccessibleDays
	AccessStartTime
	AccessEndTime
	AccessTimezone
	MinPwdLenEnforced
	MinPwdLen
	MaxFailedLoginsEnforced
	MaxFailedLogins
	MaxConcWebSessions
	MaxConcWebSessionsEnforced
	MaxConcWebSessionsUnlimited
	MaxConcWebSessionsDisplaced
Domain	DomainName*
	Description
SSO Credentials	UserName*
	ResourceName*
	ResourceType*
	ResourceUser*
	ResourcePassword*
SSO Resource	SSOResourceName*
	Description
SSO Resource Group	SSOResourceGroupName*
	Description

Table 21. Attributes by Entry Type in AddOnly Mode (continued)

Entry Type	Attribute
	SSOResources (Multivalued attribute)

The Connector does not support duplicate or multiple entries. Only one entry should be supplied to the Connector at a time.

Update Mode

When deployed in **Update** mode, the Connector is able to modify existing data in the Tivoli Access Manager database. The Connector should be added to the **Flow** section of a Tivoli Directory Integrator AssemblyLine. The **Output Map** must define a mapping for the following attributes. These attributes can be also be retrieved through querying the Connector Schema.

When importing users/groups during an update:

- The Tivoli Access Manager account can only be imported from the registry if the account does not already exist in Tivoli Access Manager.
- When the **ReplaceUsersOnUpdate** or **ReplaceroupsOnUpdate** flags (see "Connector Input Attribute Details" on page 273) are set to 'true', the user will be added as a member of the group, but an exception will be thrown if the user is already a member.
- The **description** attribute is the only attribute that will undergo an update during an import. No other Tivoli Access Manager attributes imported from the registry will be modified.

Keep in mind the caveats on manipulating Policy items and their required Policy Entry attributes as stipulated in Table 20 on page 266.

Attributes marked with an asterisk (*) are mandatory.

Table 22. Attributes by Entry Type in Update Mode

Entry Type	Attribute
User	UserName*
	Description
	Password
	IsAccountValid
	IsPasswordValid
	IsSSOUser
	MaxFailedLogins
	MaxConcWebSessions
	Groups (Multivalued attribute)
Group	GroupName*
	Description
	ReplaceUsersOnUpdate
	Users (Multivalued attribute)
Policy	UserName*
	AcctExpDateEnforced
	AcctExpDateUnlimited
	AcctExpDate

Table 22. Attributes by Entry Type in Update Mode (continued)

Entry Type	Attribute
	AcctDisableTimeEnforced
	AcctDisableTimeUnlimited
	AcctDisableTimeInterval
	PwdSpacesAllowedEnforced
	PwdSpacesAllowed
	MaxPwdAgeEnforced
	MaxPwdAge
	MaxPwdRepCharsEnforced
	MaxPwdRepChars
	MinPwdAlphas
	MinPwdAlphasEnforced
	MinPwdNonAlphasEnforced
	MinPwdNonAlphas
	TodAccessEnforced
	AccessEndTime
	AccessibleDays
	AccessStartTime
	AccessTimezone
	MinPwdLenEnforced
	MinPwdLen
	MaxFailedLoginsEnforced
	MaxFailedLogins
	MaxConcWebSessions
	MaxConcWebSessionsEnforced
	MaxConcWebSessionsUnlimited
	MaxConcWebSessionsDisplaced
Domain	DomainName*
	Description
SSO Credentials	UserName*
	ResourceName*
	ResourceType*
	ResourceUser
	ResourcePassword
SSO Resource	Not Supported
SSO Resource Group	SSOResourceGroupName*
	SSOResources (Multivalued attribute)

Additionally, any mandatory fields mentioned above should be defined in the **Link Criteria** of the Connector. The Link Criteria is required by the AssemblyLine, since the AssemblyLine will invoke the Connectors `findEntry()` method to verify the existence of the given user. The value of the attribute, as defined in the Link Criteria, must match the value of the element present in the **Output Map**.

The only operator supported for Link Criteria is an **equals exact match**. Wildcard search criteria are not supported. The Connector does not support duplicate or multiple entries. Only one entry should be supplied to the Connector at a time.

Delete Mode

When deployed in **Delete** mode, the Connector is able to delete existing data from the Tivoli Access Manager database. The Connector should be added to the **Flow** section of an AssemblyLine.

Attributes marked with an asterisk (*) are mandatory.

Table 23. Attributes by Entry Type in Delete Mode

Entry Type	Attribute
User	UserName*
Group	GroupName*
Policy	UserName*
Domain	DomainName*
SSO Credentials	UserName*
	ResourceName*
	ResourceType*
SSO Resource	SSOResourceName*
SSO Resource Group	SSOResourceGroupName*

The mandatory attribute must be defined in the **Link Criteria** of the Connector. The Link Criteria is required by the AssemblyLine, since the AssemblyLine will invoke the Connector's `findEntry()` method to verify the existence of the given user.

The only operator supported for Link Criteria is an **equals exact match**. Wildcard search criteria are not supported. The Connector does not support duplicate or multiple entries. Only one entry should be supplied to the Connector at a time.

Lookup Mode

When deployed in **Lookup** mode, the Connector is able to obtain all details of the required Tivoli Access Manager data. The Connector should be added to the **Flow** section of an AssemblyLine. The mandatory attribute must be defined in the **Link Criteria** of the Connector.

Attributes marked with an asterisk (*) are mandatory.

Table 24. Attributes by Entry Type in Lookup Mode

Entry Type	Attribute
User	UserName*
Group	GroupName*
Policy	UserName*
Domain	DomainName*
SSO Credentials	UserName*
	ResourceName*
	ResourceType*
SSO Resource	SSOResourceName*
SSO Resource Group	SSOResourceGroupName*

The Connector's `findEntry()` method is the main code executed. The only operator supported for **Link Criteria** is an equals exact match. Wildcard search criteria are not supported.

The Connector does not support duplicate or multiple entries. The Connector will return only one entry at a time.

Iterator Mode

When deployed in **Iterator** mode, the Connector is able to retrieve the details of each data entry in the Tivoli Access Manager database, in turn, and make those details available to the AssemblyLine.

When deployed in this mode, the Tivoli Directory Integrator AssemblyLine will first call the Connector's `selectEntries()` method to obtain and cache a list of all data entries in the Tivoli Access Manager database. If the entry Type is **User** or **Group** and a filter attribute was provided, then the list will contain the filtered entries. The Assembly Line will then call the Connector's `getNextEntry()` method. This method will maintain a pointer to the current name cached in the list.

Wildcards are supported for the filter attribute of **User** and **Group** entry types only:

- Asterisks can be used to create a **UserName** wildcard search pattern. The **UserName** pattern is interpreted as a string of characters that matches zero or more characters of the User's **UserName** attribute. Asterisks can be located at the beginning, in the middle or at the end of the **UserName** pattern, and the **UserName** can contain multiple asterisks.
- Asterisks can be used to create a **GroupName** wildcard search pattern. The **GroupName** pattern is interpreted as a string of characters that matches zero or more characters of the Groups's **GroupName** attribute. Asterisks can be located at the beginning, in the middle or at the end of the **GroupName** pattern, and the **GroupName** can contain multiple asterisks.

Troubleshooting

Problems may be experienced for any of the following reasons:

TAM Connector not installed properly
Check the configuration and re-configure if necessary.

Query Schema Issues

When performing a schema query using the Connectors with the Tivoli Directory Integrator GUI, an attempt to connect to the data source may result in an exception. These exceptions can be ignored. Any subsequent use of the **Discover** schema button will succeed. The Connectors do not support the *Get Next Entry* style of schema query. The Connectors do support the normal Tivoli Directory Integrator style of schema discovery.

Changing Mode of Connectors Already in AssemblyLine

During testing, it was observed that changing the mode of Connector in the AssemblyLine did not always work. The Connector sometimes appeared to execute in its original mode, resulting in AssemblyLine errors. If this occurs, delete the Connector and add it to the AssemblyLine in the new mode.

Connector Input Attribute Details

This section details the attributes for connector input.

User

Table 25. Connector Input Attributes

Attribute	Description	Example	Default
UserName	The User Name	maryl	
RegistryUID	The LDAP User Distinguished Name (DN)	cn=mary ,o=companyabc, c=au	
FirstName	The User's First Name	Mary	
LastName	The User's Last Name	Lou	
Description	A Description	Contractor	
Password	User's password (If the 'NoPasswordPolicyOnCreate' attribute is set to FALSE, the password must conform to the current password policy in Tivoli Access Manager.)	m3ry10u	
IsAccountValid	TRUE to activate the account. FALSE to leave the account inactive.	TRUE or FALSE	TRUE
IsPasswordValid	Set to FALSE if user is to change the password on next login. TRUE to remain unchanged.	TRUE or FALSE	TRUE
IsPDUser	TAM PD User flag.	TRUE or FALSE	
IsSSOUser	TRUE to enable Single Sign-on capabilities for this user. FALSE to disable.	TRUE or FALSE	FALSE
NoPasswordPolicy OnCreate	FALSE will enforce the password policy on the "Password" attribute and as a result it will be checked against the password policy settings the first time it is created. TRUE will not enforce the password policy on the password when it is created.	TRUE or FALSE	TRUE
MaxFailedLogins	Set the maximum number of failed logins a user can have before the account is disabled.	8	10
MaxConcWebSessions	Set the maximum number of concurrent web sessions allowed	3	0

Table 25. Connector Input Attributes (continued)

Attribute	Description	Example	Default
Groups (Multivalued attribute)	This is a multi-valued attribute. Please refer to the *IBM Tivoli Directory Integrator V7.1 Users Guide* about how to set multi-valued attributes. Any Group listed in this attribute should already exist as a valid group in Tivoli Access Manager.	Groups1 -> itSpecialists Groups2 -> programmers	
ReplaceGroupsOnUpdate	In Update mode, if this attribute is set to TRUE, the user is removed as a member of all of the groups with which the user is currently a member. The user is then added as members of the each of the groups supplied as values in the Groups attribute. If this attribute is set to FALSE, then during modification the groups that currently contain the user are modified to add or delete that user in accordance with each of the Groups attribute value's operation. As a result, if the Groups attribute value operation is set to AttributeValue.AV_ADD, the user will be added to the group. If the Group attribute value operation is set to AttributeValue.AV_DELETE, the user will be removed from the group. The ReplaceGroupsOnUpdate flag is ignored in Add mode. The flag is also ignored in Update mode if the update reverts to an Add operation when the user is not found to be a Tivoli Access Manager user.	TRUE or FALSE	TRUE

Group

Table 26. Group Attributes

Attribute	Description	Example
GroupName	The Group Name	programmers
RegistryGID	The LDAP Group DistinguishedName (DN)	cn=programmers, cn=SecurityGroups, secAuthority=Default
CommonName	The LDAP Common Name (CN)	programmers
Description	The Group Description	Fulltime Programmers
IsPDGroup	TAM PD Group Flag.	TRUE or FALSE
ObjectContainer	TAM Object Container	
Users	This is a multi-valued attribute. Please refer to the *IBM Tivoli Directory Integrator V7.1 Users Guide* about how to set multi-valued attributes. Any user listed in this attribute should already exist as a valid user in Tivoli Access Manager.	Users1 -> maryl Users2 -> johnd

Table 26. Group Attributes (continued)

Attribute	Description	Example
ReplaceUsersOnUpdate	In update mode, this Attribute provides a boolean flag to indicate how the membership of the group modified. If it is set to TRUE, all members of the group are removed and the list of users supplied as values in the Users attribute replaces the removed users. If this Attribute is set to FALSE, then during modification, the users of the group are modified in accordance with the User attribute value's operation. As a result, if the User attribute value operation is set to AttributeValue.AV_ADD, the user will be added as a member of the group. If the User attribute value operation is set to AttributeValue.AV_DELETE, the user will be deleted from the group's membership. The default value is TRUE. The ReplaceUsersOnUpdate flag is ignored in Add mode. The flag is also ignored in Update mode if the update reverts to an Add operation when the group is not found to be a Tivoli Access Manager group.	TRUE or FALSE

Policy

Table 27. Policy Attributes

Attribute	Description	Example
UserName	The User Name the policy will be set for. Must be a valid Tivoli Access Manager user.	maryl
AcctExpDateEnforced	If TRUE then enforce the Account Expiration Date.	TRUE or FALSE
AcctExpDateUnlimited	If TRUE then set the Account Expiration Date to be unlimited.	TRUE or FALSE
AcctExpDate	Sets the expiry date for the user account The attribute must be of type java.util.Date, or java.lang.String. If a String value is provided the required date string format is "yyyyMMdd" where 'yyyy' us the four digit year, 'MM' is the two digit month, and 'dd' is the two digit day; i.e. 20091231 is the value for the date 31st December 2009.	Refer to the Tivoli Access Manager Java API Reference.
AcctDisableTimeEnforced	If TRUE then enforce the Account Disable Time.	TRUE or FALSE
AcctDisableTimeUnlimited	If TRUE then set the Account Disable Time to be unlimited.	TRUE or FALSE
AcctDisableTimeInterval	Set the Account Disable Time Interval.	Refer to the Tivoli Access Manager Java API Reference.
PwdSpacesAllowedEnforced	If TRUE enforce the value of the 'PwdSpacesAllowed' attribute.	TRUE or FALSE

Table 27. Policy Attributes (continued)

Attribute	Description	Example
PwdSpacesAllowed	If TRUE allow spaces in the password.	TRUE or FALSE
MaxPwdAgeEnforced	If TRUE enforce the Maximum Password Age value.	TRUE or FALSE
MaxPwdAge	Sets the Maximum Password Age.	Refer to the Tivoli Access Manager Java API Reference.
MaxPwdRepCharsEnforced	If TRUE enforce the Maximum Password Repeatable characters number.	TRUE or FALSE
MaxPwdRepChars	Sets the Maximum Password Repeatable Characters.	5
MinPwdAlphasEnforced	If TRUE enforce the Minimum number of Alphanumeric characters allowed.	TRUE or FALSE
MinPwdAlphas	Sets the Minimum number of Alphanumeric characters allowed.	6
MinPwdNonAlphasEnforced	If TRUE enforce the Minimum number of non-alphanumeric characters allowed.	TRUE or FALSE
MinPwdNonAlphas	Sets the Minimum number of non-alphanumeric characters allowed.	3
TodAccessEnforced	If TRUE enforce the access times set for the user.	TRUE or FALSE
AccessibleDays	Sets the days accessible for the user account.	Refer to the Tivoli Access Manager Java API Reference.
AccessStartTime	Sets the access start time for the user account.	Refer to the Tivoli Access Manager Java API Reference.
AccessEndTime	Sets the access end time for the user account.	Refer to the Tivoli Access Manager Java API Reference.
AccessTimezone	Sets the time zone for the user account.	Refer to the Tivoli Access Manager Java API Reference.
MinPwdLenEnforced	If TRUE enforce the Minimum Password Length.	TRUE or FALSE
MinPwdLen	Sets the Minimum Password Length.	8
MaxFailedLoginsEnforced	If TRUE then enforce the Maximum Failed Login setting.	TRUE or FALSE
MaxFailedLogins	Sets the Maximum Failed Logins for the user.	8
MaxConcWebSessions	Set the maximum number of concurrent web sessions allowed.	3
MaxConcWebSessionsEnforced.	If TRUE then enforce the Maximum Concurrent Web Sessions setting.	TRUE or FALSE
MaxConcWebSessionsUnlimited	If TRUE then the maximum concurrent web sessions policy is set to "unlimited".	TRUE or FALSE
MaxConcWebSessionsDisplaced	If TRUE then the maximum concurrent web sessions policy is set to "displace".	TRUE or FALSE

Domain

Table 28. Domain Attributes

Attribute	Description	Example
DomainName	The name of the domain	MyDomain

Table 28. Domain Attributes (continued)

Attribute	Description	Example
Description	The Domain description	Sample domain name

SSO Credentials

Table 29. SSO Credentials Attributes

Attribute	Description	Example
UserName	The name of the user the credentials will be set for	maryl
ResourceName	The SSO Resource Name. (Must be a valid Tivoli Access Manager SSO Resource entry).	myResource1
ResourceType	Specifies whether this resource is a single resource or a resource group	"Web Resource" and "Resource Group" are the only allowable values.
ResourceUser	Sets the Resource User Name	marylou
ResourcePassword	Sets the User Name Password for the specified resource	b1ddy4

SSO Resource

Table 30. SSO Resource Attributes

Attribute	Description	Example
SSOResourceName	The Single sign-on Resource Name	MyResource1
Description	The Description	Development Server 1

SSO Resource Group

Table 31. SSO Resource Group Attributes

Attribute	Description	Example
SSOResourceGroupName	The Single sign-on Resource Group Name	MyResourceGroup1
Description	The Description	All Development Servers
SSOResources	This is a multi-valued attribute. Please refer to the *IBM Tivoli Directory Integrator V7.1 Users Guide* about how to set multi-valued attributes. Any SSO Resources listed in this attribute should already exist as a valid SSO Resource in Tivoli Access Manager.	SSOResources1 -> myResource1 SSOResources2 -> myResource2

See also

Access Manager for e-business

TCP Connector

The TCP Connector is a transport Connector using TCP sockets for transport. You can use the TCP Connector in Iterator and AddOnly mode only.

Iterator Mode

When in Iterator mode, the TCP Connector waits for incoming TCP calls on a specific port. When a connection is established, the **getnext** method returns an entry with the following properties:

socket The TCP socket object (for example, the TCP input and output streams)

in An instance of a BufferedReader using the socket's input stream

out An instance of a BufferedWriter using the socket's output stream

The **in** and **out** objects can be used to read and write data to or from the TCP connection. For example, you can do the following to implement a simple echo server (put the code in the **After GetNext** Hook):

```
var ins = conn.getProperty("in");
var outs = conn.getProperty("out");
var str = ins.readLine();
outs.write("You said==>"+str+"<==");
outs.flush();
```

Because you are using a BufferedWriter, it is important to call the out.flush() method to make sure data is actually sent out over the connection.

If you specify a Parser, then the BufferedReader is passed to the Parser, which in turn reads and interprets data sent on the stream. The returned entry then includes any attributes assigned by the Parser as well as the properties listed previously (**socket**, **in**, and **out**).

If the TCP Connector is configured in *Listen Mode=true* then the connection is closed between each call to the *getnext* method. If *Listen Mode=false* the connection to the remote host is kept open for as long as the TCP Connector is active (for example, until the AssemblyLine stops).

Note: The Listen Mode parameter in this connector should not be confused with the behavior of the "TCP Server Connector" on page 281, which is a connector more suited for accepting incoming (multiple concurrent ones, if necessary) TCP requests. The functionality associated with *Listen Mode=true* is deprecated and will be removed in future versions of the connector, and it will be possible to configure and use the connector for outgoing connections only.

AddOnly Mode

When the TCP Connector works in this mode, the default implementation is to write entries in their string form, which is not useful. Typically, you specify a Parser or use the **Override Add** hook to preform specific output. In the **Override Add** hook you access the **in** or **out** objects by calling the Connector Interface's getReader() and getWriter() methods, for example:

```
var in = mytcpconnector.connector.getReader();
var out = mytcpconnector.connector.getWriter();
```

You can also use the **Before Add** and **After Add** hooks to insert headers or footers around the output from your Parser.

Configuration

TCP Host
The remote host to which connections are made (**servermode = false**).

TCP Port
The TCP port number to connect or listen to (depends on the value of **servermode**).

Use SSL
If checked, the Connector will deploy the Secure Socket Layer (SSL) on the connection.

Listen Mode
(Deprecated, use TCP Server Connector instead) If **true**, then Iterating listens for incoming requests. If **false**, then Iterating connects to a remote server.

Need Client Authentication over SSL
(Deprecated) If checked and if SSL is enabled in Listen Mode (that is, listening for incoming connections), client authentication is necessary.

Connection Backlog
(Deprecated) This represents the maximum queue length for incoming connection indications (a request to connect). If a connection indication arrives when the queue is full, the connection is refused.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

You can select a Parser for this Connector from the Parser pane, where you select a parser by clicking the top-left **Select Parser** button.

See also

"File system Connector" on page 95,
"Direct TCP /URL scripting" on page 41,
"TCP Server Connector" on page 281
"URL Connector" on page 285.

TCP Server Connector

This Connector supports Server and Iterator modes only.

In Server mode, this Connector waits for incoming TCP connections on a specified port and spawns a new thread to handle the incoming request. When the new thread has started, the original Server mode Connector goes back to listening mode. When the newly created thread has completed, the thread stops and the TCP connection is closed.

In Iterator mode, the Connector is single-threaded, in that it waits for a connection on the IP address of the local machine and the port specified. Once the connection is received, the Connector will generate Entries based on received data until the Client closes the connection.

Configuration

TCP Port
The TCP port on which to listen for incoming connections.

Connection Backlog
This represents the maximum queue length for incoming connection indications (a request to connect). If a connection indication arrives when the queue is full, the connection is refused.

Use SSL
If checked, the Connector will deploy the Secure Socket Layer (SSL) on the connection.

Require Client Authentication
If checked, the Connector will require clients to supply client-side SSL certificates that can be matched to the configured Tivoli Directory Integrator trust store.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Connector Schema

The Connector makes the following properties available in the Input Attribute Map:

tcp.originator
The Connector object.

event.originator
The Connector object. This is the same object as the one stored in **tcp.originator**. This Attribute ensures backward compatibility with the now obsolete TCP Port EventHandler.

tcp.inputstream
TCP socket input stream (java.io.InputStream)

event.inputstream
TCP socket input stream (java.io.InputStream). This is the same object as the one stored in **tcp.inputstream**. This Attribute ensures backward compatibility with the obsolete TCP Port EventHandler.

tcp.outputstream
TCP socket output stream (java.io.OutputStream).

event.outputstream
TCP socket output stream (java.io.OutputStream). This is the same object as the one stored in **tcp.outputstream**. This Attribute ensures backward compatibility with the obsolete TCP Port EventHandler.

tcp.remoteIP
Remote IP address (dot notation).

tcp.remotePort
Remote TCP port number.

tcp.remoteHost
Remote hostname.

tcp.localIP
Local IP address (dot notation).

tcp.localPort
Local TCP port number.

tcp.localHost
Local hostname.

tcp.socket
TCP Socket object (java.net.Socket).

The TCP Server Connector does not use its Output Attribute Map – it just closes the Connection to the client application when done.

The **tcp.inputstream** and **tcp.outputstream** Attribute values are meant to be used via scripting in the AssemblyLine to read the client request and write the response respectively.

See also

"TCP Connector" on page 279.

Timer Connector

The timer waits for a specified time; then it returns from sleep and resumes an AssemblyLine, that is, it starts a new cycle. This Connector runs in Iterator mode only.

On attribute mapping, there is one attribute you can map into the work entry: a timestamp, which is of type `java.util.Date`. It will contain the time when it started the cycle.

Using Delta functionality with this Connector does probably not make much sense.

Configuration

This Connector needs the following parameters:

Month
Select a month to run the Timer Connector (* = any)

Day Day of the month to run the Timer Connector on (* = any)

Weekday
Select a weekday to run the Timer Connector (* = any)

Hours The hour(s) at which to run the Timer Connector (* = any). You can specify multiple values in a comma-separated list, but the values must be in ascending order.

Minutes
The minute(s) at which to run the Timer Connector. You can specify multiple values in a comma-separated list, but the values must be in ascending order.

Schedule
This is a UNIX crontab-style **schedule** parameter to set up when to run the Connector; when this parameter is specified it overrides all other timing parameters. The main use of this would be to specify more than one weekday to run; for example to specify that the Connector should be run at 03:45 every weekday, the schedule parameter could be set to "* * 2,3,4,5,6 3 45".

Detailed Log
If this field is checked, additional log messages are generated.

Comment
This parameter can hold any user comments. It is not taken into account during the operation of the Connector.

URL Connector

The URL Connector is a transport Connector that requires a Parser to operate. The Connector opens a stream specified by a URL.

Note: When forced through a firewall that enforces a proxy server, the URL Connector does not work. The URL Connector needs to have the right proxy server set.

This Connector supports AddOnly and Iterator modes.

The Connector, in principle, can handle secure communications using the SSL protocol, but it may require driver–specific configuration steps in order to set up SSL support.

Configuration

The Connector needs the following parameters:

URL The URL to open (for example, http://*host*/file.csv).

Detailed Log
If this parameter is checked, more detailed log messages are generated.

From the Parser configuration pane, you can select a Parser to operate upon the stream. You select a parser by clicking on the bottom-right **Inherit from:** button.

Supported URL protocol

The supported URL protocols are:

- HTTP
- HTTPS

See also

"File system Connector" on page 95,
"TCP Connector" on page 279,
"Direct TCP /URL scripting" on page 41.

Web Service Receiver Server Connector

The Web Service Receiver Server Connector is part of the Tivoli Directory Integrator Web Services suite.

Note: Due to limitations of the Axis library used by this component only WSDL (http://www.w3.org/TR/wsdl) version 1.1 documents are supported. Furthermore, the supported message exchange protocol is SOAP 1.1.

This Connector is basically an HTTP Server specialized for servicing SOAP requests over HTTP. It operates in Server mode only.

AssemblyLines support an Operation Entry (op-entry). The op-entry has an attribute *$operation* that contains the name of the current operation executed by the AssemblyLine. In order to process different web service operations easier, the Web Service Receiver Server Connector will set the *$operation* attribute of the op-entry.

The Web Service Receiver Server Connector supports generation of a WSDL file according to the input and output schema of the AssemblyLine. As in Tivoli Directory Integrator 7.1 AssemblyLines support multiple operations, the WSDL generation can result in a web service definition with multiple operations. There are some rules about naming the operations:

- Pre-6.1 TDI configuration files contain only one input and one output schema referred to as default operation schemas. When a pre-6.1 TDI configuration is used the only operation generated is named as the name of the AssemblyLine as in TDI 6.0.
- In Tivoli Directory Integrator 7.1 configurations if there is an operation named "Default", the corresponding operation in the WSDL file is named as the name of the AssemblyLine.
- In Tivoli Directory Integrator 7.1 configurations if there is an operation named "Default" and there is also an operation with a name as the name of the AssemblyLine, both operations preserve their names in the WSDL file.
- In all other cases the operations appear in the WSDL file as they are named in the AssemblyLine configuration.

Hosting a WSDL file

The Web Service Receiver Server Connector provides the "*wsdlRequested*" Connector Attribute to the AssemblyLine.

If an HTTP request arrives and the requested HTTP resource ends with "?WSDL" then the Connector sets the value of the "*wsdlRequested*" Attribute to **true**; otherwise the value of this Attribute is set to **false**.

This Attribute's value tells the AssemblyLine whether the request received is a SOAP request or a request for a WSDL file, and allows the AssemblyLine to distinguish between pure SOAP requests and HTTP requests for the WSDL file. The AssemblyLine can use a branch component to execute only the required piece of logic – (1) when a request for the WSDL file has been received, then the AssemblyLine can read a WSDL file and send it back to the web service client; (2) when a SOAP request has been received the AssemblyLine will handle the SOAP request. Alternatively, you could program the `system.skipEntry();` call at an appropriate place (in a script component, in a hook in the first Connector in the AssemblyLine, etc.) to skip further processing.

It is the responsibility of the AssemblyLine to provide the necessary response to either a SOAP request or a request for a WSDL file.

The Connector implements a public method:

```
public String readFile (String aFileName) throws IOException;
```

This method can be used from Tivoli Directory Integrator JavaScript in a script component to read the contents of a WSDL file on the local file system. The AssemblyLine can then return the contents of the WSDL in the "*soapResponse*" Attribute, and thus to the web service client in case a request for the WSDL was received.

Schema

Input Schema

Table 32. Web Service Receiver Server Connector Input Schema

Attribute	Value
host	Type is String. Contains the name of the host to which the request is sent. This parameter is set only if "wsdlRequested" is false.
requestedResource	The requested HTTP resource.
soapAction	The SOAP action HTTP header. This parameter is set only if "wsdlRequested" is false.
soapRequest	The SOAP request in txt/XML or DOMElement format. This parameter is set only if "wsdlRequested" is false.
wsdlRequested	This parameter is true if a WSDL file is requested and false otherwise.
http.username	This attribute is used only when HTTP basic authentication is enabled. The value is the username of the client connected.
http.password	This attribute is used only when HTTP basic authentication is enabled. The value is the password of the client connected.

Output Schema

Table 33. Web Service Receiver Server Connector Output Schema

Attribute	Value
responseContentType	The response type. The default response type is "text/xml". It is used with SOAP messages.
soapResponse	The SOAP response message. If wsdlRequested is true, then soapResponse is set to the contents of the WSDL file.
http.credentialsValid	This attribute is used only when HTTP basic authentication is enabled. When the client provides username and password for HTTP Basic Authentication then this attribute must be set to true or false (this is not done by the Connector, it's done by the AssemblyLine using the Connector). If the value is true this means that the client is authenticated correctly and access is granted. If the value is false then the user is not authenticated and an HTTP "Not Authorized" Connector response is returned.

Configuration

Parameters

TCP Port
The port number the service is running (listening) on.

Connection Backlog
This represents the maximum queue length for incoming connection indications (a request to connect). If a connection indication arrives when the queue is full, the connection is refused.

Input the SOAP message as
Specifies the type of the SOAP Request message input to the AssemblyLine. This drop-down list allows you to choose either **String** or *"DOMElement"*.

Return the SOAP message as
Specifies the type of the SOAP Response message output from the AssemblyLine. This drop-down list allows you to choose either **String** or *"DOMElement"*.

Tag Op-Entry
When this parameter is checked (that is, "true") the Connector will tag the op-entry even if the currently executed operation is not on the list of exposed operations in the AssemblyLine/WSDL. It is up to the Tivoli Directory Integrator solution implementation to handle this case appropriately.

Use SSL
If checked the server will only accept SSL (https) connections.The SSL parameters (keystore, etc.) are specified as values of Java system properties in the `global.properties` file located in the Tivoli Directory Integrator installation folder.

Require Client Authentication
Specifies whether this Connector will require clients to authenticate with client SSL certificates. If the value of this parameter is **true** (that is, checked) and the client does not authenticate with a client SSL certificate, then the Connector will drop the client connection. If the value of this parameter is **true** and the client does authenticate with a client SSL certificate, then the Connector will continue processing the client request. If the value of this parameter is **false**, then the Connector will process the client request regardless of whether the client authenticates with a client SSL certificate.

Auth Realm
The basic-realm sent to the client in case authentication is requested.

Use HTTP Basic Authentication
This connector supports HTTP basic authentication. To activate, check the "Use HTTP Basic Authentication" checkbox. If activated, the server checks if any credentials are already sent and if not, the server sends authorization request to client. After the client sends the needed credentials, the Connector then sets two attributes: "http.username" and "http.password". These two attributes contain the username and password of the client. It is responsibility of the AssemblyLine to check if this pair of username and password is valid. If the client is authorized successfully then "http.credentialsValid" work Entry Attribute must be set to true. If the client is not authorized then "http.credentialsValid" work Entry Attribute must be set to false. If the client is not authorized then the server sends a "Not Authorized" HTTP message.

Comment
Your own comments go here.

Detailed Log
If checked, will generate additional log messages.

WSDL Output to Filename
The name of the WSDL file to be generated when the **Generate WSDL** button is clicked. This parameter is only used by the WSDL Generation Utility – this parameter is not used during the Connector execution.

Web Service provider URL
The address on which web service clients will send web service requests. Also this parameter is only used by the WSDL Generation Utility – this parameter is not used during the Connector execution.

The **Generate WSDL** button runs the WSDL generation utility.

The WSDL Generation utility takes as input the name of the WSDL file to generate and the URL of the provider of the web service (the web service location). This utility extracts the input and output parameters of the AssemblyLine in which the Connector is embedded and uses that information to generate the WSDL parts of the input and output WSDL messages. It is mandatory that for each Entry

Attribute in the "Initial Work Entry" and "Result Entry" Schema the "Native Syntax" column be filled in with the Java type of the Attribute (for example, "`java.lang.String`"). The WSDL file generated by this utility can then be manually edited.

The operation style of the SOAP Operation defined in the generated WSDL is *rpc*.

The WSDL generation utility cannot generate a <types...>...</types> section for complex types in the WSDL.

Connector Operation

The Web Service Receiver Server Connector stores the following information from the HTTP/SOAP request into Attributes of the Connector's *conn* entry, ready to be mapped into the *work* entry (also see "Schema" on page 288):

- The name of the host to which the request is sent (the local host) – stored into the "*host*" Attribute
- The requested HTTP resource – stored into the "*requestedResource*" Attribute
- The value of the "*soapAction*" HTTP header – stored into the "*soapAction*" Attribute
- If the value of the **Input the SOAP message as** FC parameter is **String** then the SOAP request message is stored as a `java.lang.String` object in the "*soapRequest*" Attribute.
- If the value of the **Input the SOAP message as** FC parameter is **DOMElement** then the SOAP request message is stored as an `org.w3c.dom.Element` object in the "*soapRequest*" Attribute.
- Whether a WSDL file was requested — in the "*wsdlRequested*" Attribute. If this is the case (that is, the value is **true**, no other Attributes will be set).

When reaching the Response stage of the AssemblyLine, this Connector requires the SOAP response message in text XML form or as *DOMElement* from the "*soapResponse*" Attribute of the *work* Entry to be mapped out:

- If the value of the **Return the SOAP message as** FC parameter is **String** then the SOAP response message must be stored as a `java.lang.String object` in the "*soapResponse*" Attribute by the AssemblyLine.
- If the value of the **Return the SOAP message as** FC parameter is **DOMElement** then the SOAP response message must be stored as a `org.w3c.dom.Element` in the "*soapResponse*" Attribute by the AssemblyLine.

The Connector then wraps the SOAP response message into an HTTP response and returns it to the web service client.

See also

"Axis Easy Web Service Server Connector" on page 19.

z/OS LDAP Changelog Connector

The z/OS LDAP Changelog Connector is a specialized instance of the LDAP Connector. It is configured for usage with a z/OS Directory Server, accessed using the LDAP protocol over TCP/IP ("zLDAP").

There are some differences in the way the changes to password policy operational attributes are logged to `cn=changelog` in IBM Tivoli Directory Server on z/OS and in Distributed IBM Tivoli Directory Server (which runs on other platforms). See “Differences between changelog on distributed TDS and z/OS TDS” on page 124 for details on the currently known differences in behavior between the two versions.

This connector supports Delta Tagging, at the Entry level, the Attribute level and the Attribute Value level. It is the LDIF Parser that provides delta support at the Attribute and Attribute Value levels.

This connector is able to intercept changes from the changelog of a RACF® (Resource Access Control Facility) LDAP server. RACF is the security manager of z/OS and it maintains a database containing usernames and passwords. Changes to this database can be logged in the changelog of an LDAP server such as IBM Tivoli Directory Server (TDS). The changelog of this server can be accessed trough the GDBM LDAP interface and the RACF database itself - through the SDBM interface. This connector is suitable for propagating changes of sensitive information (usernames, passwords, and so forth) across LDAP servers on different z/OS machines or other distributed platforms.

The Connector will detect *modrdn* operations in the Server's changelog, see “Detect and handle modrdn operation” on page 182 for more information.

Note: This component is not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.

Attribute merge behavior

In older versions of Tivoli Directory Integrator, in the z/OS LDAP Changelog Connector merging occurs between Attributes of the changelog Entry and changed Attributes of the actual Directory Entry. This creates issues because you cannot detect the attributes that have changed. The Tivoli Directory Integrator 7.1 version of the Connector has logic to address these situations, configured by a parameter: **Merge Mode**. The modes are:

- **Merge changelog and changed data** - The Connector merges the attributes of the Changelog Entry with changed attributes of the actual Directory Entry. This is the older implementation and keeps backward compatibility.
- **Return only changed data** - Returns only the modified/added attributes and makes Changelog Iterator and Delta mode easier. This is the default; note that in configurations developed under and migrated from earlier versions of Tivoli Directory Integrator, you may need to select **Merge changelog and changed data** manually so as to ensure identical behavior.
- **Return both** - Returns an Entry which contains changed attributes of the actual Directory Entry and an additional attribute called "changelog" which contains attributes of the Changelog Entry. Allows you to easily distinguish between two sets of Attributes.

Delta tagging is supported in all merge modes and entries can be transferred between different LDAP servers without much scripting.

Configuration

The Connector needs the following parameters:

LDAP URL
The LDAP URL for the connection (ldap://*host*:*port*).

Login username
The LDAP distinguished name used for authentication to the server. Leave blank for anonymous access.

Login password
The credentials (password).

Authentication Method
Type of LDAP authentication. Can be one of the following:

- **Anonymous** - If this authentication method is set then the server, to which a client is connected, does not know or care who the client is. The server allows such clients to access data configured for non-authenticated users. The Connector automatically specifies this authentication method if no username is supplied. However, if this type of authentication is chosen and **Login username** and **Login password** are supplied, then the Connector automatically sets the authentication method to Simple.
- **Simple** - using **Login username** and **Login password**. Treated as anonymous if **Login username** and **Login password** are not provided. Note that the Connector sends the fully qualified distinguished name and the client password in cleartext, unless you configure the Connector to communicate with the LDAP Server using the SSL protocol.
- **CRAM-MD5** - This is one of the SASL authentication mechanisms. On connection, the LDAP Server sends some data to the LDAP client (that is, this Connector). Then the client sends an encrypted response, with password, using MD5 encryption. After that, the LDAP Server checks the password of the client. CRAM-MD5 is supported only by LDAP v3 servers. It is not supported against any supported versions of Tivoli Directory Server.
- **SASL** - The client (this Connector) will use a Simple Authentication and Security Layer (SASL) authentication method when connecting to the LDAP Server. Operational parameters for this type of authentication will need to be specified using the **Extra Provider Parameters** option; for example, in order to setup a DIGEST-MD5 authentication you will need to add the following parameter in the Extra Provider Parameters field:

  ```
  java.naming.security.authentication:DIGEST-MD5
  ```

  For more information on SASL authentication and parameters see: http://java.sun.com/products/jndi/tutorial/ldap/security/sasl.html.

  Note: Not all directory servers support all SASL mechanisms and in some cases do not have them enabled by default. Check the documentation and configuration options for the directory server you are connecting to for this information.

Use SSL
If Use SSL is **true** (that is, checked), the Connector uses SSL to connect to the LDAP server. Note that the port number might need to be changed accordingly.

ChangeLog Base
The search base where the Changelog is kept. The standard DN for this is **cn=changelog**.

Extra Provider Parameters
Allows you to pass a number of extra parameters to the JNDI layer. It is specified as name:value pairs, one pair per line.

Iterator State Key
Specifies the name of the parameter that stores the current synchronization state in the User Property Store of the IBM Tivoli Directory Integrator. This must be a unique name for all parameters stored in one instance of the IBM Tivoli Directory Integrator User Property Store.

Start at
Specifies the starting changenumber. Each Changelog entry is named **changenumber=intvalue** and the Connector starts at the number specified by this parameter and automatically increases by one. The special value **EOD** means start at the end of the Changelog.

State Key Persistence
Governs the method used for saving the Connector's state to the System Store. The default is **End of Cycle**, and choices are:

After read
Updates the System Store when you read an entry from the directory server's change log, before you continue with the rest of the AssemblyLine.

End of cycle
Updates the System Store with the change log number when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual
Switches off the automatic updating of the System Store with this Connector's state information; instead, you will need to save the state by manually calling the z/OS LDAP Changelog Connector's *saveStateKey()* method, somewhere in your AssemblyLine.

Merge mode
Governs the method used for merging attributes of the Changelog Entry and changed attributes of the actual Directory Entry. The default is **Return only changed data**. The possible values are:

- **Merge changelog and changed data** - Pre-7.0 implementation; for backward compatibility.
- **Return only changed data** - Returns only the modified/added attributes.
- **Return both** - Returns changed attributes of the actual Directory Entry, plus an additional attribute called "changelog" that contains an Entry with changelog attributes.

Timeout
Specifies the number of seconds the Connector waits for the next Changelog entry. The default is 0, which means wait forever.

Sleep Interval
Specifies the number of seconds the Connector sleeps between each poll. The default is 60.

Detailed Log
If this field is checked, additional log messages are generated.

Note: Changing Timeout or Sleep Interval values will automatically adjust its peer to a valid value after being changed (for example, when timeout is greater than sleep interval the value that was not edited is adjusted to be in line with the other). Adjustment is done when the field editor loses focus.

See also

Change logging in Tivoli Directory Server for z/OS,
Accessing RACF Resource Profiles through the IBM Tivoli Directory Server for z/OS,
RACF Documentation,
“LDAP Connector” on page 181,
“Active Directory Change Detection Connector” on page 8,
“IBM Tivoli Directory Server Changelog Connector” on page 123,
“Sun Directory Change Detection Connector” on page 215.

Chapter 3. Parsers

Parsers are used in conjunction with a stream-based Connector to interpret or generate the content that travels over the Connector's byte stream.

Parsers cooperate with their calling Connectors in discovering the schema of the underlying data stream when you press **Discover Schema**.

When the bytestream you are trying to parse is not in harmony with the chosen Parser, you get a `sun.io.MalformedInputException` error. For example, the error message can show up when using the **Schema** tab to browse a file.

Base Parsers

- "CBE Parser" on page 297
- "CSV Parser" on page 301
- "DSML Parser" on page 303
- "DSMLv2 Parser" on page 305
- "Fixed Parser" on page 317
- "HTTP Parser" on page 319
- "LDIF Parser" on page 325
- "Line Reader Parser" on page 329
- "Script Parser" on page 331
- "Simple Parser" on page 335
- "SOAP Parser" on page 337
- "SPMLv2 Parser" on page 339
- "Simple XML Parser" on page 347
- "XML Parser" on page 351
- "XML SAX Parser" on page 361
- "XSL based XML Parser" on page 365

Character Encoding conversion

IBM Tivoli Directory Integrator is written in Java which in turn supports Unicode (double byte) character sets. When you work with strings and characters in AssemblyLines and Connectors, they are always assumed to be in Unicode. Most Connectors provide some means of Character Encoding to be used. When you read from text files on the local system, Java has already established a default Character Encoding conversion that is dependent on the platform you are running.

The Tivoli Directory Integrator Server has the **-n** command line option, which specifies the character set of Config files it will use when writing new ones; it also embeds this character set designator in the file so that it can correctly interpret the file when reading it back in later.

However, occasionally you read or write data from or to text files in which information is encoded in different Character Encodings (this could happen if you are reading a file created on a machine running a different operating system). The Connectors that require a Parser usually accept a **Character Set** parameter in the Parser configuration. If set, this parameter must be set to one of the accepted conversion

tables found in the Java runtime, as governed by the IANA Charset Registry. If this parameter is not set, most Parsers use the local character set. Some Parsers might have specific default character sets. See information about individual Parsers in this guide.

Some files, when UTF-8, UTF-16 or UTF-32 encoded, may contain a Byte Order Marker (BOM) at the beginning of the file. The purpose of the BOM is to help finding the algorithm used for encoding the InputStream to characters. A BOM is the encoding of the character `0xFEFF`. This can be used as a signature for the encoding used. The Tivoli Directory Integrator File Connector does not recognize a BOM. Also, these Tivoli Directory Integrator Parsers do not recognize a BOM:

- CSVParser
- FixedParser
- HTTPParser
- LDIFParser
- LineReaderParser
- ScriptParser
- SimpleParser

If you try to read a file with a BOM, and the Parser does not know how to handle this, then in order to avoid returning unusable data, you should add this code to, for example, the **Before Selection** Hook of the connector:

```
var bom = thisConnector.connector.getParser().getReader().read(); // skip the BOM = 65279
if (bom != -1 && bom != 65279) {
//make sure that we are skipping the BOM and not any other meaningful character.
 throw "Invalid BOM";
}
```

This code will read and skip the BOM, assuming that you have specified the correct character set for the parser. This workaround is only needed if the Parser does not recognize or process the BOM, or a skip of the BOM is needed in general.

Some care must be taken with the HTTP protocol; see “HTTP Parser” on page 319, section “Character sets/Encoding” on page 324 about character sets encoding in the description of the HTTP Parser for more details.

Availability

Please refer to the IANA Charset Registry (http://www.iana.org/assignments/character-sets).

A common character set on Windows computers is CP850; for i5/OS a common value is IBM037.

CBE Parser

This Parser extends the “XML Parser” on page 351, and is designed to read XML from the input stream and convert this XML to a CBE object; alternatively to write XML based on CBE objects as attributes provided. It operates similar to the “CBE Function Component” on page 395, except it is now packaged as a Parser. This means it can be attached to an input and/or output stream as part of a Connector's configuration instead of having to be deployed separately as another component in an AssemblyLine flow.

For example, it is possible you might want to save in a file all CBE objects received in some AssemblyLine; you could use a “File system Connector” on page 95 in AddOnly mode with the CBE Parser and pass the CBE object itself to the "event" attribute of the Output Map.

You can read those CBE objects back in again using a File system Connector in Iterator mode with the CBE Parser configured, and you will receive the objects in the "event" attribute of the Input Map.

Using the Parser

When the CBE Parser reads from XML it returns all standard CBE attributes and the CBE object as attribute of the Input Map.

When the CBE Parser writes to XML it expects the CBE object to be set to the "event" attribute of the Output Map. In case the "event" is not set the CBE Parser expects that at least all the required CBE attributes are set in the Output Map otherwise an error will be thrown.

The XML passed to this parser should comply with the CBE specification and the CBE schema.

CBE Input and Output Map Attributes

The CBE specification has a complex structure. To enable you to define the attributes, a dotted notation for the attributes is supported in the attribute map for some of the components. To know more about these attributes and their suggested values please refer to the Autonomic Computing Toolkit Developer's Guide at: http://www-128.ibm.com/developerworks/autonomic/books/fpy0mst.htm#ToC_91.

The following CBE Attributes list are the attributes that are used to create a CBE event object or the attributes that are going to be filled out from a CBE object provided to the event attribute or to the eventXml attribute of the Output Map.

Table 34. CBE attributes

Attribute Name	Attribute Type	Description
CreationTime	String	The time the event was created. Default value will be the time when the event object is created in the CBEGeneratorFC.
GlobalInstanceId	String	Primary Identifier of event. A unique id will be generated if not passed by user.
Message	String	Optional property
Severity	Integer	Optional property. Value ranges from 0 – 70
ExtensionName	String	Optional property.
SequenceNumber	Integer	Optional property
RepeatCount	Integer	Optional property.
ElapsedTime	Integer	Optional property if RepeatCount is not set.
Priority	Integer	Optional property. Values range from 0 – 100.
situation.		***This notation defines properties of the situation object.***

Table 34. CBE attributes (continued)

Attribute Name	Attribute Type	Description
situation.CategoryName	String	Describes the type of Situation. Defined Values are • StartSituation • StopSituation • ConnectSituation • ConfigureSituation • RequestSituation • FeatureSituation • DependencySituation • CreateSituationDestroy • SituationReportSituation • AvailableSituation • OtherSituation This is a required property.
situation.reasoningScope		Describes scope of the situation. This is a required property.
availableSituation.operationDisposition	String	Required if CategoryName is AvailableSituation
availableSituation.processingDisposition	String	Required if CategoryName is AvailableSituation
availableSituation.availabilityDisposition	String	Required if CategoryName is AvailableSituation
configureSituation.successDisposition	String	Required if CategoryName is ConfigureSituation
connectSituation.successDisposition	String	Required if CategoryName is ConnectSituation
connectSituation.situationDisposition	String	Required if CategoryName is ConnectSituation
createSituation.successDisposition	String	Required if CategoryName is CreateSituation
dependencySituation.dependencyDisposition	String	Required if CategoryName is DependencySituation
destroySituation.successDisposition	String	Required if CategoryName is DestroySituation
featureSituation.featureDisposition	String	Required if CategoryName is FeatureSituation
reportSituation.reportCategory	String	Required if CategoryName is ReportSituation
requestSituation.successDisposition	String	Required if CategoryName is RequestSituation
requestSituation.situationQualifier	String	Required if CategoryName is RequestSituation
startSituation.successDisposition	String	Required if CategoryName is StartSituation
startSituation.situationQualifier	String	Required if CategoryName is StartSituation
stopSituation.successDisposition	String	Required if CategoryName is StopSituation
stopSituation.situationQualifier	String	Required if CategoryName is StopSituation
otherSituation.any	String	Required if CategoryName is OtherSituation **Note:** The value must be represented as an XML element in order for the XML to comply with the CBE 1.0.1 Schema (for example, <someTag> Any Text Value Inside </someTag>). The value should be wrapped in only one element (for example, this is not valid value: <someTag>val</someTag> <anotherTag>val2</anotherTag>).
SCI.		***This notation describes the source component identification. These are required properties for a CommonBaseEvent.***

Table 34. CBE attributes (continued)

Attribute Name	Attribute Type	Description
SCI.location	String	
SCI.locationType	String	
SCI.executionEnvironment	String	
SCI.component	String	
SCI.subcomponent	String	
SCI.componentIdType	String	
SCI.componentType	String	
RCI.		***This notation describes component identification information for the reporter component. This is not a required property.***
RCI.location	String	
RCI.locationType	String	
RCI.executionEnvironment	String	
RCI.component	String	
RCI.subcomponent	String	
RCI.componentIdType	String	
RCI.componentType	String	
X.		***This notation describes an ExtendedDataElement (EDE). This is not a required property.***
X.attributeName	String	Creates EDE with name `attribute Name` and value defined by user
X.attributeName.childAttributeName	String	Creates EDE with name `attribute Name` and child element with name `childAttributeName`

When working with CBE objects using either the CBE FC or CBE Parser in an AssemblyLine, you will need to deal with Input and Output maps.

Table 35. Input map attributes

Attribute Name	Attribute Type	Description
Event	org.eclipse.hyades.logging.events.cbe.CommonBaseEvent	The CBE object converted from the attributes.
eventXml	String	The XML representation of the converted CBE object.
Including all the CBE attributes from table Table 34 on page 297.		

Table 36. Output map attributes

Attribute Name	Attribute Type	Description
Event	org.eclipse.hyades.logging.events.cbe.CommonBaseEvent	The CBE object that is going to be converted to attributes.
eventXml	String	The XML that is going to be parsed to CBE object and then converted to attributes.
Including all the CBE attributes from table Table 34 on page 297.		

Configuration

The CBE Parser uses the following parameters:

Character Encoding
Sets Character encoding to use when reading or writing. The default value is UTF-8.

When reading XML, this parameter will be used only if the input source does not already have encoding defined.

Since this parser extends the XML Parser, the same considerations as for that Parser apply.

Omit XML Declaration
When set, this causes the Parser to omit an XML declaration header in the output stream.

Validate XML
When set, this causes the Parser to validate the read XML with the XSD schema from the CBE specification.

Detailed Log
Checking this causes the Parser to output additional log messages.

See also

"CBE Function Component" on page 395,

"XML Parser" on page 351,

"Simple XML Parser" on page 347.

CSV Parser

The Comma Separated Values (CSV) Parser reads and writes data in a CSV format.

Note: In the Config Editor, the parameters are set in the **Parser** tab of the Connector. If you want to use TAB as a Field Separator you need to specify **\t**, but when supplying Field Names you must use the actual tab character between field names.

On output, multi-valued attributes only deliver their first value.

Configuration

The Parser has the following parameters:

Field Separator
Specifies the character used to separate each column; typically a comma or semicolon. If not specified, the parser attempts to guess when reading, and uses a comma when writing. You can use backslash (\ ) as the escape character to specify non-printable characters. For example, (\t) denotes the TAB character.

Sort Fields
Check this option to write header fields in alphabetical (ascending) order. The default is false, that is unchecked.

Field Names
Specifies the name for each column the parser must read or write. If not specified, the parser reads the first line and uses the value as field names. You can use the Field Separator between the field names, or specify each name on a separate line.

Enable Quoting
On write, when this parameter is set to **true** (that is, checked), the field is output with quotation marks around it under the same conditions as in previous versions, however, quotation marks inside a quoted field are now doubled.

Note: If **Enable Quoting** is set to **false**, the field is output as is, which can cause problems.

When reading, quotation marks around the field are stripped if this parameter is set to **true**, and the parser is able to read quoted attributes containing the column separator. If this parameter is set to **false**, the parser returns unexpected values when the input contains fields delimited by quotation marks.

Quote all fields
Quote all fields independently if they contain quotation mark, separator or new line

Write Header
The default value for this parameter is **true**. If **Write Header** is set, the first line output by the parser contains all the field names separated by the column separator.

Log long lines
Define a maximum number of bytes for a line. Linenumbers of lines longer than this maximum number are logged.

Combine remainder in last field
if checked, combine all extra fields from lines exceeding the number of defined fields into a new "Remainder" field.

The fields, and implicitly, the number of fields, are defined either using the **Field Names** parameter, or in absence of this, the first line of the file.

Character Encoding
Character Encoding to be used. Also see “Character Encoding conversion” on page 295.

Detailed Log

If this field is checked, additional log messages are generated.

Schema

The schema which the CSV Parsers provides to the Input/Output Connector map is taken from the value of the **Field Names** configuration parameter of the Parser. The parser will simply copy the fields from the parameter to the Maps of the Connector. This saves you from copying all the fields one by one from the Parser to the corresponding Connector Map.

DSML Parser

The DSML Parser reads and writes XML documents. The Parser silently ignores schema entries.

Configuration

The Parser has the following parameters:

DN Attribute
The attribute used for the distinguished name DSML attribute (**$dn**).

DSML prefix
Prefix used on XML elements to indicate that they belong to the DSML namespace. Default is **dsml**.

DSML namespace URI
The URI which identifies this namespace. Default is http://www.dsml.org/DSML.

Omit XML Declaration
If checked, the XML declaration is omitted in the output stream.

Document Validation
If checked, this parser requests a DTD/Schema-validating parser.

Namespace Aware
If checked, this parser requests a namespace-aware parser.

Character Encoding
Character Encoding to be used.

This Parser extends the Simple XML Parser; therefore, the same notices with regards to Character Encoding apply.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Examples

The following example shows how you can generate DSML documents dynamically:

```
var dsml = system.getParser ( "ibmdi.DSML" );
var entry = system.newEntry();

entry.setAttribute ("$dn", "uid=johnd,o=doe.com");
entry.setAttribute ("mail", "john@doe.com");
entry.setAttribute ("uid", "johnd");
entry.setAttribute ("objectclass", "top");
entry.addAttributeValue ("objectclass", "person");

dsml.setOutputStream ( new java.io.StringWriter() );
// Uncomment if you dont want the "<?xml version= ...." header
// dsml.setOmitXMLDeclaration ( true );
dsml.initParser();
dsml.writeEntry ( entry );
dsml.closeParser();

var result = dsml.getXML();
task.logmsg ( result );
```

The following example shows how you can read a DSML document using script:

```
var dsml = system.getParser ("ibmdi.DSML");
dsml.setInputStream ( new java.io.FileInputStream("dirdata.dsml") );
dsml.initParser ();
```

```
var entry = dsml.readEntry();
while ( entry != null ) {
 task.dumpEntry ( entry );
 entry = dsml.readEntry();
}
```

See also

"Simple XML Parser" on page 347,
"SOAP Parser" on page 337,
"DSMLv2 Parser" on page 305.

DSMLv2 Parser

The Directory Services Markup Language v1.0 (DSMLv1) enables the representation of directory structural information as an XML document. DSMLv2 goes further, providing a method for expressing directory queries and updates (and the results of these operations) as XML documents. DSMLv2 documents can be used in a variety of ways. IBM Tivoli Directory Integrator provides a Parser that can parse and create DSMLv2 request and response messages.

The DSMLv2 Parser is initialized with a DSMLv2 batch request or DSMLv2 batch response. Individual calls to read or write Entries will result in parsing or creation of individual DSML requests or responses (as parts of the batch request or response).

The Parser supports Delta tagging at the Entry level and the Attribute level. See also "Multiple Attribute modifications" on page 312.

Modes

The DSMLv2 Parser operates either in Server or in Client mode:

- In Server mode the Parser reads and parses DSMLv2 requests; and writes and creates DSMLv2 responses
- In Client mode the Parser reads and parses DSMLv2 responses; and writes and creates DSMLv2 requests.

Operations

The DSMLv2 Parser supports **Modify**, **Add**, **Delete**, **Search**, **ModifyDN**, **Compare**, **Auth** and **Extended** operations.

Attention: The following Tivoli Directory Integrator 6.0 DSMLv2 Parser custom helper objects from the ITIM DSML library are no longer supported:

- dsml.request – for all request operations.
- dsml.response – for all response operations.

If you have configurations using either of these Attributes, you must edit the configurations to remove any reference to these Attributes. The data available through the raw request and response objects in older versions are not available through the other Attributes delivered by the DSMLv2 Parser.

Modify Request

Entries with the following structure are parsed (on read) and created (on write) by the parser for Modify Requests:

Table 37.

Attribute	Value
dsml.operation	set to "modifyRequest"
dsml.base	holds the "dn" XML attribute of the DSML "modifyRequest" element
$dn	holds the "dn" XML attribute of the DSML "modifyRequest" element

Additionally, for each modification item: a Tivoli Directory Integrator attribute named as the "name" XML attribute of the DSML "modification" element, with the values specified for the "modification" DSML element and Tivoli Directory Integrator attribute's operation set as the "operation" XML attribute of the DSML "modification" element.

Modify Response

Entries with the following structure are parsed (on read) and created (on write) by the parser for Modify Responses:

Table 38.

Attribute	Value
dsml.operation	modifyResponse
$dn	holds the "matchedDN" XML attribute of the DSML "modifyResponse" element
dsml.resultcode	holds the "code" XML attribute of the "resultCode" XML element of the DSML response
dsml.resultdescr	holds the "descr" XML attribute of the "resultCode" XML element of the DSML response
dsml.error	the presence of this attribute indicates an error condition and holds the value of the "errorMessage" XML element of the DSML response
dsml.exception	holds a javax.naming.NamingException object that is used to automatically fill in the "code" and "descr" XML attributes of the "resultCode" XML element of the DSML response; if this attribute is specified any values set to the "dsml.resultcode" and "dsml.resultdescr" Entry Attributes are ignored and replaced with data retrieved through the exception object.
dsml.referral	holds a vector containing all referral URIs of the DSML "addResponse" element

Search Request

Entries with the following structure are parsed (on read) and created (on write) by the parser for Search Requests:

Table 39.

Attribute	Value
dsml.operation	set to "searchRequest"
$dn	holds the "matchedDN" XML attribute of the DSML "compareResponse" element
dsml.base	holds the "dn" XML attribute of the DSML "searchRequest" element
dsml.scope	holds the value of the "scope" attribute of the DSML "searchRequest" element
dsml.filter	the LDAP filter that corresponds to the "filter" element of the DSML request
dsml.attributes	the value of this attribute is a Vector or String whose elements hold the names of the attributes listed in the "attributes" element of the DSML request.
dsml.derefAliases	holds the value of the "derefAliases" attribute of the DSML "searchRequest" element
dsml.sizeLimit	holds the value of the "sizeLimit" attribute of the DSML "searchRequest" element
dsml.timeLimit	holds the value of the "timeLimit" attribute of the DSML "searchRequest" element
dsml.typesOnly	holds the value of the "typesOnly" attribute of the DSML "searchRequest" element

Search Response

Entries with the following structure are parsed (on read) and created (on write) by the parser for Search Responses:

Table 40.

Attribute	Value
dsml.operation	set to "searchResponse"
$dn	holds the "matchedDN" XML attribute of the DSML "searchResultDone" element of the DSML response
dsml.resultcode	holds the "code" XML attribute of the "resultCode" XML element of the DSML response

Table 40. (continued)

Attribute	Value
dsml.resultdescr	holds the "descr" XML attribute of the "resultCode" XML element of the DSML response
resultEntries	a multi-valued attribute, each of its values is a Tivoli Directory Integrator Entry whose attributes correspond to the "attr" elements of the corresponding "searchResultEntry" element.
dsml.error	the presence of this attribute indicates an error condition and holds the value of the "errorMessage" XML element of the DSML response
dsml.exception	holds a javax.naming.NamingException object that is used to automatically fill in the "code" and "descr" XML attributes of the "resultCode" XML element of the DSML response; if this attribute is specified, any values set to the "dsml.resultcode" and "dsml.resultdescr" Entry Attributes are ignored and replaced with data retrieved through the exception object.

Add Request

Entries with the following structure are parsed (on read) and created (on write) by the parser for Add Requests:

Table 41.

Attribute	Value
dsml.operation	set to "addRequest"
dsml.base	holds the "dn" XML attribute of the DSML "addRequest" element
$dn	holds the "dn" XML attribute of the DSML "addRequest" element

Additionally, for each DSML attr element: a Tivoli Directory Integrator Attribute named as the "name" XML attribute of the DSML "attr" element and as values specified for the "attr" DSML element.

Add Response

Entries with the following structure are parsed (on read) and created (on write) by the parser for Add Responses:

Table 42.

Attribute	Value
dsml.operation	set to "addResponse"
"$dn	holds the "matchedDN" XML attribute of the DSML "addResponse" element
dsml.resultcode	holds the "code" XML attribute of the "resultCode" XML element of the DSML response
dsml.resultdescr	holds the "descr" XML attribute of the "resultCode" XML element of the DSML response
dsml.error	the presence of this attribute indicates an error condition and holds the value of the "errorMessage" XML element of the DSML response
dsml.exception	holds a javax.naming.NamingException object that is used to automatically fill in the "code" and "descr" XML attributes of the "resultCode" XML element of the DSML response; if this attribute is specified, any values set to the "dsml.resultcode" and "dsml.resultdescr" Entry Attributes are ignored and replaced with data retrieved through the exception object.
dsml.referral	holds a vector containing all referral URIs of the DSML "addResponse" element

Delete Request

Entries with the following structure are parsed (on read) and created (on write) by the parser for Delete Requests:

Table 43.

Attribute	Value
dsml.operation	set to "deleteRequest"
dsml.base	holds the "dn" XML attribute of the DSML "delRequest" element
$dn	holds the "dn" XML attribute of the DSML "delRequest" element

Delete Response

Entries with the following structure are parsed (on read) and created (on write) by the parser for Delete Responses:

Table 44.

Attribute	Value
dsml.operation	set to "deleteResponse"
$dn	holds the "matchedDN" XML attribute of the DSML "delRequest" element
dsml.resultcode	holds the "code" XML attribute of the "resultCode" XML element of the DSML response
dsml.resultdescr	holds the "descr" XML attribute of the "resultCode" XML element of the DSML response
dsml.error	the presence of this attribute indicates an error condition and holds the value of the "errorMessage" XML element of the DSML response
dsml.exception	holds a javax.naming.NamingException object that is used to automatically fill in the "code" and "descr" XML attributes of the "resultCode" XML element of the DSML response; if this attribute is specified, any values set to the "dsml.resultcode" and "dsml.resultdescr" Entry Attributes are ignored and replaced with data retrieved through the exception object.
dsml.referral	holds a vector containing all referral URIs of the DSML "addResponse" element

ModifyDN Request

Entries with the following structure are parsed (on read) and created (on write) by the parser for ModifyDN Requests:

Table 45.

Attribute	Value
dsml.operation	set to "modDnRequest"
dsml.base	holds the "dn" XML attribute of the DSML "modDNRequest" element
$dn	holds the "dn" XML attribute of the DSML "modDNRequest" element
newrdn	holds the "newrdn" XML attribute of the DSML "modDNRequest" element
dsml.newSuperior	holds the "newSuperior" XML attribute of the DSML "modDNRequest" element
dsml.deleteOldRDN	holds the "deleteoldrdn" XML attribute of the DSML "modDNRequest" element

ModifyDN Response

Entries with the following structure are parsed (on read) and created (on write) by the parser for ModifyDN Responses:

Table 46.

Attribute	Value
dsml.operation	set to "modDnResponse"
$dn	holds the "matchedDN" XML attribute of the DSML "modDNResponse" element
dsml.resultcode	holds the "code" XML attribute of the "resultCode" XML element of the DSML response
dsml.resultdescr	holds the "descr" XML attribute of the "resultCode" XML element of the DSML response
dsml.error	the presence of this attribute indicates an error condition and holds the value of the "errorMessage" XML element of the DSML response
dsml.exception	holds a javax.naming.NamingException object that is used to automatically fill in the "code" and "descr" XML attributes of the "resultCode" XML element of the DSML response; if this attribute is specified, any values set to the "dsml.resultcode" and "dsml.resultdescr" Entry Attributes are ignored and replaced with data retrieved through the exception object.
dsml.referral	holds a vector containing all referral URIs of the DSML "addResponse" element

Compare Request

Entries with the following structure are parsed (on read) and created (on write) by the parser for Compare Requests:

Table 47.

Attribute	Value
dsml.operation	set to "compareRequest"
dsml.base	holds the "dn" XML attribute of the DSML "compareRequest" element
$dn	holds the "dn" XML attribute of the DSML "compareRequest" element
dsml.compare_name	holds the "name" XML attribute of the "assertion" element of the DSML request
dsml.compare_value	holds the value of the "assertion" element of the DSML request

Compare Response

Entries with the following structure are parsed (on read) and created (on write) by the parser for Compare Responses:

Table 48.

Attribute	Value
dsml.operation	set to "compareResponse"
$dn	holds the "matchedDN" XML attribute of the DSML "compareResponse" element
dsml.compare_result	either "true" of "false" depending on whether the compare found match or not. When the Parser is used to create a DSML response, this attribute is required and depending on its value the Parser sets the right result code value.
dsml.resultcode	holds the "code" XML attribute of the "resultCode" XML element of the DSML response
dsml.resultdescr	holds the "descr" XML attribute of the "resultCode" XML element of the DSML response
dsml.error	the presence of this attribute indicates an error condition and holds the value of the "errorMessage" XML element of the DSML response

Table 48. (continued)

Attribute	Value
dsml.exception	holds a javax.naming.NamingException object that is used to automatically fill in the "code" and "descr" XML attributes of the "resultCode" XML element of the DSML response; if this attribute is specified, any values set to the "dsml.resultcode" and "dsml.resultdescr" Entry Attributes are ignored and replaced with data retrieved through the exception object.
dsml.referral	holds a vector containing all referral URIs of the DSML "addResponse" element

Auth Request

Entries with the following structure are parsed (on read) and created (on write) by the parser for Auth Requests:

Table 49.

Attribute	Value
dsml.operation	set to "authRequest"
dsml.principal	holds the "principal" XML attribute of the DSML "authRequest" element

Auth Response

Entries with the following structure are parsed (on read) and created (on write) by the parser for Auth Responses:

Table 50.

Attribute	Value
dsml.operation	set to " authResponse"
$dn	holds the "matchedDN" XML attribute of the DSML "authResponse" element
dsml.resultcode	holds the "code" XML attribute of the "resultCode" XML element of the DSML response
dsml.resultdescr	holds the "descr" XML attribute of the "resultCode" XML element of the DSML response
dsml.error	the presence of this attribute indicates an error condition and holds the value of the "errorMessage" XML element of the DSML response.
dsml.exception	holds a javax.naming.NamingException object that is used to automatically fill in the "code" and "descr" XML attributes of the "resultCode" XML element of the DSML response; if this attribute is specified, any values set to the "dsml.resultcode" and "dsml.resultdescr" Entry Attributes are ignored and replaced with data retrieved through the exception object.
dsml.referral	holds a vector containing all referral URIs of the DSML "authResponse" element

Extended Request

Entries with the following structure are parsed (on read) and created (on write) by the parser for Extended Requests:

Table 51.

Attribute	Value
dsml.operation	set to "extendedRequest"
dsml.extended.requestname	holds the "requestName" XML attribute of the DSML "extendedRequest" element
dsml.extended.requestvalue	holds the "requestValue" XML attribute of the DSML "extendedRequest" element

Extended Response

Entries with the following structure are parsed (on read) and created (on write) by the parser for Extended Response:

Table 52.

Attribute	Value
dsml.operation	set to "extendedResponse"
$dn	holds the "matchedDN" XML attribute of the DSML "extendedResponse" element
dsml.resultcode	holds the "code" XML attribute of the "resultCode" XML element of the DSML response
dsml.resultdescr	holds the "descr" XML attribute of the "resultCode" XML element of the DSML response
dsml.error	the presence of this attribute indicates an error condition and holds the value of the "errorMessage" XML element of the DSML response
dsml.exception	holds a javax.naming.NamingException object that is used to automatically fill in the "code" and "descr" XML attributes of the "resultCode" XML element of the DSML response; if this attribute is specified, any values set to the "dsml.resultcode" and "dsml.resultdescr" Entry Attributes are ignored and replaced with data retrieved through the exception object.
dsml.referral	holds a vector containing all referral URIs of the DSML "extendedResponse" element
dsml.responseName	holds the "responseName" XML attribute of the DSML "extendedResponse" element
dsml.response	holds byte array containing string which represents the response from an "extendedResponse" operation

Note: All invalid XML characters (as per the XML specification) are removed from the "dsml.error" Entry Attribute before serializing this attribute into DSML.

Error Response

Entries with the following structure are parsed (on read) and created (on write) by the parser for Error Response:

Table 53.

Attribute	Value
dsml.operation	set to "errorResponse"
dsml.errorType	holds the value of the "type" XML attribute of the DSML response XML element; must be one of "notAttempted", "couldNotConnect", "connectionClosed", "malformedRequest", "gatewayInternalError", "authenticationFailed", "unresolvableURI" or "other"
dsml.message	holds the text value of the "message" XML element of the DSML response
dsml.details	holds the value of the "detail" XML element of the DSML response

Binary and non-String Attributes

When parsing DSML messages, attributes tagged as binary by the **Binary Attributes** Parser parameter are Base64 decoded, that is, the string value from the DSML message is Base64 decoded to Java byte array.

When creating DSML messages, all Attributes whose value is Java byte array are Base64 encoded to String before being written in the DSML message.

If when creating a DSML message an Attribute is passed whose value's type is neither String nor Java byte array, the value is converted to String by calling the object's "toString()" method and this String value is written in the DSML message.

Optional Attributes

The following optional attributes, when present, are parsed (on read) and created (on write) by the parser for all DSMLv2 Requests and Responses:

Table 54.

Attribute	Value
dsml.requestID	corresponds to the DSMLv2 "requestID" attribute.
dsml.controls	holds Vector of raw `com.ibm.ldap.dsml.DsmlControl` objects.

DSMLv2 controls must be Base64 encoded

When reading, the Parser expects the values of DSMLv2 controls to be Base64 encoded. For example instead of a control element like this one:

```
<control type="1.2.840.113556.1.4.619" criticality="true">
 <controlValue xsi:type="xsd:string">mycontrolvalue</controlValue>
</control>
```

you need to provide a control element like the following:

```
<control type="1.2.840.113556.1.4.619" criticality="true">
 <controlValue xsi:type="xsd:base64Binary">bXljb250cm9sdmFsdWU=</controlValue>
</control>
```

This is a limitation of the underlying DSML library from IBM Directory Server (com.ibm.ldap.dsml.*). The DSMLv2 XML Schema (http://www.oasis-open.org/committees/dsml/docs/DSMLv2.xsd) allows controlValue elements to be of xsd:anyType. However, the IBM DS DSMLv2 library (obtained from TDS 6.1) ignores the xsi:type attribute and always attempts to base64 decode the value.

Setting result code and result description

When setting the "dsml.resultcode" Attribute for DSML Response messages, allowed types are: java.lang.Integer and java.lang.String containing an integer value as string. This value corresponds to the integer "code" XML attribute of the "resultCode" DSML element and it is required by the DSMLv2 specification.

You can optionally set the "dsml.resultdescr" Attribute for DSML Response messages. This value corresponds to the "descr" XML attribute of the "resultCode" DSML element. It is not required by the DSMLv2 specification. When you assign a value to this attribute it is placed in the DSML response as is – no validation of the value (which is an enumerated string is done) and no check is performed whether this value corresponds to the mandatory integer "dsml.resultcode" Attribute.

The "code" and "descr" XML attributes of the "resultCode" DSML element can also be set through the "dsml.exception" Entry Attribute for DSML Response messages. This attribute can only accept javax.naming. NamingException objects. When "dsml.exception" attribute is set, the "code" and "descr" XML attributes of the "resultCode" DSML element are overwritten with new values extracted from the exception object. For example when the "dsml.exception" attribute is set to a javax.naming.AuthenticationException object, the "code" attribute will be set to the LDAP code of "49" and the "descr" attribute will be set to the LDAP description "inappropriateAuthentication".

Multiple Attribute modifications

The DSMLv2 Parser (and LDIF Parser) does not support multiple modifications over a single Attribute – the values from a modification are accumulated in the Attribute and the operation from the last

modification is set as the operation tag for the Attribute. Therefore, the Parsers need to merge the modifications in an Entry in such way that the resulting Attribute modification be equivalent to the modifications for that Attribute in the modify operation. This can be achieved by using `Attribute.ATTRIBUTE_MOD` – a Tivoli Directory Integrator-specific tagging at the Attribute level and by using AttributeValue level tagging - `AttributeValue.AV_ADD`, `AttributeValue.AV_DELETE`.

The following data flow rules will be used when accumulating modifications in an Attribute object:

- On modification "Add" – the value(s) will be added with `AttributeValue.AV_ADD` to the Attribute; also the Attribute will be tagged as `Attribute.ATTRIBUTE_MOD` unless it is already tagged as `Attribute.ATTRIBUTE_REPLACE`
- If the Attribute is already tagged with `Attribute.ATTRIBUTE_REPLACE` in a previous modification this tag will not be changed
- On modification "Delete" with value(s) – the value(s) will be added with `AttributeValue.AV_DELETE` to the Attribute; also the Attribute will be tagged as `Attribute.ATTRIBUTE_MOD` unless it is already tagged as `Attribute.ATTRIBUTE_REPLACE`; if the Attribute is tagged as `Attribute.ATTRIBUTE_REPLACE` for each value in the "Delete" modification the value will be removed from the Attribute if that value is present in the Attribute
- On modification "Delete" without values – the Attribute values from previous modifications will be cleared and the Attribute will be tagged as `Attribute.ATTRIBUTE_REPLACE`
- On modification "Replace" – the Attribute values from previous modifications will be cleared and the new ones will be added; also the Attribute will be tagged as `Attribute.ATTRIBUTE_REPLACE`.

Configuration

The Parser needs the following parameters:

Character Encoding
Specifies the XML character encoding; for example, UTF-8 or ASCII.

This Parser extends the Simple XML Parser; therefore, the same notices with regards to Character Encoding apply.

Mode Specifies whether the Parser operates in Server or in Client mode – possible values are "Server" and "Client". In Server mode, requests are read and responses are written. In Client mode, requests are written and responses are read.

Binary Attributes

Specifies a comma delimited list of attributes that will be treated by the Parser as binary attributes.

The following attributes are specified as binary by default (but you can change this list):

- photo
- personalSignature
- audio
- jpegPhoto
- javaSerializedData
- thumbnailPhoto
- thumbnailLogo
- userPassword
- userCertificate
- authorityRevocationList
- certificateRevocationList
- crossCertificatePair
- x500UniqueIdentifier

- objectGUID
- objectSid

On Error
A BatchRequest element can contain the XML-attribute `onError`, which determines how the server responds to failures while processing request elements. The valid values are: `exit` and `resume`. The default value is `exit`.

Processing
Sets the value of the "processing" DSML attribute for Batch Requests.

Response Order
Influences how the server orders individual responses within the BatchResponse. The values of this parameter are `sequential` and `unordered`. The default value is `sequential`. If the Response Order value is set to `sequential`, the server must return a BatchResponse in which the individual responses maintain a positional correspondence with the individual requests.

Omit XML Declaration
Determines whether XML declaration omitting is enabled or disabled. By default, this parameter is disabled.

Indent Output
If checked, the output will be indented according to the depth of the statement lines. This is cosmetic only; it has no bearing upon the semantic content of the output file.

Soap Binding
When turned on, the parser processed and creates SOAP DSML message. Otherwise the DSML messages are not wrapped in SOAP.

Detailed Log
If checked, more detailed log messages will be generated.

Examples

Parsing a DSMLv2 AddRequest in Server mode

If the DSMLv2 Parser is configured to run in "server" (read) mode and is passed the following DSMLv2 request:

```
<batchRequest onError="exit" processing="sequential"
    responseOrder="sequential" xmlns="urn:oasis:names:tc:DSML:2:0:core"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <addRequest requestID = "3" dn="cn=chavdar kovachev,o=ibm,c=us">
        <attr name="objectclass">
            <value>person</value>
        </attr>
        <attr name="telephoneNumber">
            <value>555</value>
        </attr>
        <attr name="sn">
            <value>kovachev</value>
        </attr>
        <attr name="cn">
            <value>chavdar kovachev</value>
        </attr>
    </addRequest>
</batchRequest>
```

it will generate an Entry object with the following Attributes:

- sn: 'kovachev'
- $dn: 'cn=chavdar kovachev,o=ibm,c=us'
- telephoneNumber: '555'

- objectclass: 'person'
- dsml.operation: 'addRequest'
- dsml.requestID: '3'
- cn: 'chavdar kovachev'
- dsml.base: 'cn=chavdar kovachev,o=ibm,c=us'

Creating a DSMLv2 SearchRequest in Client mode

If the DSMLv2 Parser is configured to run in "client" (write) mode and is passed an Entry with the following Attributes:

- $dn: "o=ibm,c=us"
- dsml.derefAliases: 'neverDerefAliases'
- dsml.sizeLimit: '0'
- dsml.operation: 'searchRequest'
- dsml.timeLimit: '0'
- dsml.typesOnly: 'false'
- dsml.requestID: '7'
- dsml.attributes: '[cn, sn]'
- dsml.scope: 'wholeSubtree'
- dsml.base: 'o=ibm,c=us'
- dsml.filter: '(sn=*)'

it will generate the following DSMLv2 request:

```
<?xml version="1.0" encoding="UTF-8"?>
<batchRequest onError="exit" processing="sequential"
    responseOrder="sequential" xmlns="urn:oasis:names:tc:DSML:2:0:core"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <searchRequest requestID="7" derefAliases="neverDerefAliases"
        dn="o=ibm,c=us" scope="wholeSubtree" sizeLimit="0"
        timeLimit="0" typesOnly="false">
       <filter>
            <present name="sn"/>
       </filter>
       <attributes>
            <attribute name="cn"/>
            <attribute name="sn"/>
       </attributes>
    </searchRequest>
</batchRequest>
```

Fixed Parser

The Fixed Parser reads and writes fixed length text records.

Configuration

The Parser has the following parameters:

Column Description
This multi-line parameter specifies each field name, the offset and length. For example:

```
field1, 1, 12
field2, 13, 4
field3, 17, 3
```

These field names will show up when Schema discovery is performed.

Note: Offsets start at 1; invalid values like 0 may cause an exception.

Trim values
Check to remove leading and trailing spaces from fields read.

Character Encoding
Character Encoding to be used. Also see "Character Encoding conversion" on page 295.

Detailed Log
If this field is checked, additional log messages are generated.

HTTP Parser

The HTTP Parser interprets a byte stream according to the HTTP specification. This Parser is used by the HTTP Client Connector and by the HTTP Server Connector.

Configuration

The Parser has the following parameters:

Headers As Properties
If set, the header values are **retrieved as Properties** and **set as Properties**. If not set, the header values are **read as Attributes** and **returned as Attributes**.

Client Mode
If set, the parser operates in client HTTP response mode. If not set, the parser operates in server mode. This is of interest only if the Parser is writing an output stream.

Character Encoding
Character Encoding to be used. Also see "Character sets/Encoding" on page 324.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Schema

The HTTP Parser sets the following Attributes in the *work* Entry (Input Attribute Map and Output Attribute Map). Note that when configuration parameter **Header as Properties** is enabled this schema is not useful because all attributes described below will be configured as Entry properties.

http.method
The method to be performed on the resource identified by the Request-URI. The method is case-sensitive (default is **GET**). See http://www.w3.org/Protocols/HTTP/Methods.html for more information about HTTP methods.

http.base
URI which identifies the resource upon which to apply the request.

http.responseCode
A 3-digit integer result code of the attempt to understand and satisfy the request. This attribute or property is mandatory in client mode.

http.responseMsg
Short textual description of the Response Code. This attribute or property is mandatory in client mode.

http.body
Body of the message. Used to carry the entity-body associated with the request or response message. The message-body differs from the entity-body only when a transfer-coding has been applied, as indicated by the **http.Transfer-Encoding** header field. When reading, depending on the content-type of the data, this object is an instance of java.lang.StringBuffer, a char[] or a byte[].

http.url
The URL to use. This attribute or property is mandatory in client mode.

http.remote_user
Username if present in **http.Authorization** header field of request message.

http.remote_pass
Password if present in **http.Authorization** header field of request message.

http.status
Used when writing in server mode. The default is **200 OK**. Used to compose the Status-Line of

the HTTP response message (see http://tools.ietf.org/html/rfc2616#section-6.1). Must contain the HTTP response Status-Code (3 digit number) and the HTTP response Reason-Phrase separated by a single space character. For example "201 Created". As an alternative you can use one of the following predefined values:

- **OK** or **200 OK** - Returns a 200 OK response.
- **FORBIDDEN** or **401 Forbidden** - Returns a 401 Forbidden response. The response uses the **http.auth-realm** attribute or property.
- **NOT FOUND** or **404 File Not Found** - Returns a 404 File Not Found response.

http.auth-realm
Used when requesting additional authentication. The default value is **IBM-Directory-Integrator**.

http.redirect
When this attribute or property has a value, and you are writing and in server mode, redirect message pointing to the value of this attribute or property is sent.

http.qs.*
Parts of the query string when reading in server mode. The key is the part of the name after **http.qs**. The value is contained in the attribute or property.

http.* All other attributes or properties beginning with **http.** are used to generate a header line when writing. When reading, headers are put into attributes or properties with a name beginning with **http.**, and continuing with the name of the header.

General Header fields

http.Cache-Control
Used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain.

http.Connection
Allows the sender to specify options that are desired for that particular connection and MUST NOT be communicated by proxies over further connections.

http.Date
Represents the date and time at which the message was originated. The field value is an HTTP-date and has following format: 1*2DIGIT month 2*4DIGIT.

http.Pragma
Used to include implementation-specific directives that might apply to any recipient along the request/response chain. All pragma directives specify optional behavior from the viewpoint of the protocol.

http.Trailer
Indicates that the given set of header fields is present in the trailer of a message encoded with chunked transfer-coding.

http.Transfer-Encoding
Indicates what (if any) type of transformation has been applied to the message body in order to safely transfer it between the sender and the recipient. This differs from the content-coding in that the transfer-coding is a property of the message, not of the entity.

http.Upgrade
Allows the client to specify what additional communication protocols it supports and would like to use if the server finds it appropriate to switch protocols. This field is used within a 101 code (Switching Protocols).

http.Via
Allows the client to specify what additional communication protocols it supports and would like to use if the server finds it appropriate to switch protocols. This field is used within a 101 code (Switching Protocols).

http.Warning
Used to carry additional information about the status or transformation of a message which might not be reflected in the message. It has this format: `3DIGIT-warn-code SP warn-agent SP warn-text [SP warn-date]`.

Entity Header Fields

http.Allow
Lists the set of methods supported by the resource identified by the Request-URI. The purpose of this field is strictly to inform the recipient of valid methods associated with the resource. An Allow header field is present in a 405 (Method Not Allowed) response.

http.content-encoding
Used as a modifier to the media-type. When present, its value indicates what additional content codings have been applied to the entity-body, and thus what decoding mechanisms must be applied in order to obtain the media-type referenced by the http.Content-Type field.

http.Content-Language
Describes the natural language(s) of the intended audience for the enclosed entity. Note that this might not be equivalent to all the languages used within the entity-body.

http.content-length
Indicates the size of the entity-body, in decimal number of OCTETs, sent to the recipient or, in the case of the HEAD method, the size of the entity-body that would have been sent if the request was a GET. This attribute or property is returned when reading, and ignored when writing. It is recomputed by the Parser.

http.Content-Location
MAY be used to supply the resource location for the entity enclosed in the message when that entity is accessible from a location separate from the requested resource's URI. (absolute URI or relative URI).

http.Content-MD5
Is an MD5 digest of the entity-body for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body.

http.Content-Range
Sent with a partial entity-body to specify where in the full entity-body the partial body should be applied.

http.content-type
Indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had if the request was a GET.

http.Expires
Gives the date/time after which the response is considered stale.

http.Last-Modified
Indicates the date and time at which the origin server believes the variant was last modified. The format is HTTP-date.

Request Header Fields

http.Accept
Used to specify a set of desired media types which are acceptable for the response.

http.Accept-Charset
Used to indicate what character sets are acceptable for the response.

http.Accept-Encoding
Used to specify content-codings that are acceptable in the response.

http.Accept-Language
Used to specify set of natural languages that are preferred as a response to the request.

http.authorization
Consists of credentials containing the authentication information of the user agent for the realm of the resource being requested.

http.Expect
Used to indicate that particular server behaviors are required by the client.

http.From
If given, it contains an Internet e-mail address for the human user who controls the requesting user agent.

http.Host
Specifies the Internet host and port number of the resource being requested, as obtained from the original URI given by the user or referring resource (generally an HTTP URL). The Host field value MUST represent the naming authority of the origin server or gateway given by the original URL.

http.If-Match
Used with a method to make it conditional. A client that has one or more entities previously obtained from the resource can verify that one of those entities is current by including a list of their associated entity tags in the If-Match header field. The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead. It is also used, on updating requests, to prevent inadvertent modification of the wrong version of a resource. As a special case, the value "*" matches any current entity of the resource.

http.If-Modified-Since
Used with a method to make it conditional: if the requested variant has not been modified since the time specified in this field, an entity will not be returned from the server; instead, a 304 (not modified) response will be returned without any message-body. The format is HTTP-date.

http.If-None-Match
Used with a method to make it conditional. A client that has one or more entities previously obtained from the resource can verify that none of those entities is current by including a list of their associated entity tags in the If-None-Match header field.

http.If-Range
If a client has a partial copy of an entity in its cache, and wishes to have an up-to-date copy of the entire entity in its cache, it could use the Range request-header with a conditional GET. If the requested entity is unchanged, the part(s) that the client misses are send, otherwise - entire new entity. MAY contain HTTP date.

http.If-Unmodified-Since
Used with a method to make it conditional. If the requested resource has not been modified since the time specified in this field, the server would perform the requested operation as if the If-Unmodified-Since header were not present. If the requested variant has been modified since the specified time, the server will not perform the requested operation, and will return a 412 code (Precondition Failed). The format is HTTP-date.

http.Max-Forwards
Provides a mechanism with the TRACE and OPTIONS methods to limit the number of proxies or gateways that can forward the request to the next inbound server.

http.Proxy-Authorization
Allows the client to identify itself (or its user) to a proxy which requires authentication. Consist of credentials containing the authentication information of the user agent for the proxy and/or realm of the resource being requested.

http.Range
Indicates what range(s) (in bytes) of the result entity returned from HTTP request (using GET methods) will be received.

http.Referer
Allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained.

http.TE
Indicates what extension transfer-codings it is willing to accept in the response and whether or not it is willing to accept trailer fields in a chunked transfer-coding.

http.User-Agent
Contains information about the user agent originating the request.

Response Header Fields

http.Accept-Ranges
Indicates that server accepts range requests for a resource but even if it is missing that doesn't mean not accepting.

http.Age
Conveys the sender's estimate of the amount of time since the response (or its revalidation) was generated at the originating server.

http.ETag
Provides the current value of the entity tag for the requested variant.The entity tag MAY be used for comparison with other entities from the same resource.

http.Location
Used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource. The field value consists of a single absolute URI.

http.Proxy-Authenticate
Included as part of a 407 (Proxy Authentication Required) response. The field value consists of a challenge that indicates the authentication scheme and parameters applicable to the proxy for this Request-URI.

http.Retry-After
Can be used with a 503 (Service Unavailable) response to indicate how long the service is expected to be unavailable to the requesting client. This field MAY also be used with any 3xx (Redirection) response to indicate the minimum time the user-agent is asked wait before issuing the redirected request. The value of this field can be either an HTTP-date or an integer number of seconds (in decimal) after the time of the response.

http.Server
Contains information about the software used by the originating server to handle the request.

http.Vary
Indicates the set of request-header fields that fully determines, while the response is fresh, whether a cache is permitted to use the response to reply to a subsequent request without revalidation. For uncacheable or stale responses, the Vary field value advises the user agent about the criteria that were used to select the representation

http.WWW-Authenticate
Included in 401 (Unauthorized) response messages. The field value consists of at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

Character sets/Encoding

Character set when reading

The default character encoding when reading is **iso-8859–1**. This encoding is overridden by the **Character Encoding** parameter in the config pane for this Connector; and this characterSet parameter is overridden in turn by a header of the type "`Content-type: text/plain; charset=iso-8859–1`". For optimum performance and compatibility, this header should be present.

Character set when sending

The default character encoding when reading is **iso-8859–1**. This encoding is overridden by the **Character Encoding** parameter in the config pane for this Connector. When sending a text message, the Entry to send should contain an attribute with the name "http.content-type", having a text value of the form "`Content-type: text/plain; charset=iso-8859-1`". The defaults will be used only if this attribute is not present .

If the http.body attribute is a java.io.File object, that file will be sent as is, no character conversion will be performed.

For further observations on Character Sets, also see “Character Encoding conversion” on page 295.

How to use HTTP cookies

HTTP cookies are HTTP headers whose syntax conforms to the HTTP State Management Mechanism standard (RFC 2109, RFC 2965).

The HTTP components of IBM Tivoli Directory Integrator do not perform any special processing of cookie headers. If you wish to use cookies, you have to interpret the content of each cookie header yourself.

To set a cookie in an HTTP response use the "Set-cookie" HTTP header. For example:

```
work.setAttribute("http.Set-Cookie", "myname=myvalue; expires=Sat, 15-Jan-2011 13:23:56 GMT; path=/; domain=.ibm.com");
```

To set a cookie in an HTTP request use the "Cookie" HTTP header. For example:

```
work.setAttribute("http.Cookie", "myname=myvalue; myname2=myvalue2");
```

See also

“Character Encoding conversion” on page 295
“HTTP Client Connector” on page 107,
“HTTP Server Connector” on page 115.

LDIF Parser

The LDIF format is used to convey directory information, or a description of a set of changes made to directory entries. An LDIF file consists of a series of records separated by line separators. A record consists of a sequence of lines describing a directory entry, or a sequence of lines describing a set of changes to a directory entry. An LDIF file specifies a set of directory entries, or a set of changes to be applied to directory entries, but not both.

There is a one-to-one correlation between LDAP operations that modify the directory (add, delete, modify, moddn and modrdn), and the types of changerecords described in the LDIF format ("add", "delete", "modify", "modrdn" or "moddn"). This correspondence is intentional, and permits a straightforward translation from LDIF changerecords to protocol operations.

The LDIF Parser reads and writes LDIF style data. The LDIF Parser is usually used to do file exchange with an LDAP directory.

The LDIF Parser correctly parses and writes MIME BASE64 encoded strings: it tries to perform BASE64 encoding if necessary. One such situation is where there are trailing spaces after attribute values: to make sure another LDIF Parser gets the space, it encodes the attribute as BASE64.

Note: A conforming LDIF file must always have **Character Encoding** set to UTF-8. The **Character Encoding** parameter is also applied when encoding or decoding BASE64 encoded strings.

BASE64 encoding looks like garbled text if you do not know how to decode it.

This Parser handles/provides tags compatible with Delta Tagging at the Entry level, the Attribute level and the Attribute Value level. Delta tagging at the Attribute level is handled as in the DSMLv2 Parser, see “Multiple Attribute modifications” on page 312.

The LDIF Parser detects in its writeEntry method if the "newrdn" attribute exists and if yes, it sets the changetype to "modrdn" instead of "modify". Also see “Detect and handle modrdn operation” on page 182 for information how certain Connectors handle "modrdn" operations.

Note: This component is not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.

Reading LDIF input

While reading, the lines of the input are read one by one and the following checks are made for each:

- if a "dn" key is read this key is set to the value of the configured "dnAttributeName" attribute
- if an attribute has a value which starts with ":" it is read as bytes array with the specified encoding

The Entry is Delta tagged correspondingly if a key "changetype" is found and its value equals to "modify", "moddn" or "modrdn".

The Entry’s attributes are tagged correspondingly if any of the following keys are found – "add", "replace" or "delete".

Writing LDIF output

While writing first it is checked whether the **Version Number** parameter is selected, and if yes the text "version: 1" is written on the first line. This means that the output is according to the RFC 2489 LDIF specification. After that the "dn" key is added with the value in the "dnAttributeName" attribute (if such exists).

If the entry is Delta tagged then the corresponding changetype key is added with the value "add", "modify", "modrdn" or "delete", depending on the Entry's operation and attributes. If the parameter **Only Descriptive Records** is set, however, a changerecord is not written, even if the Entry is Delta tagged.

If an Entry's attribute is Delta tagged then the corresponding operation is added in the output – "add", "replace" or "delete" with the value of the attribute.

Configuration

The Parser has the following parameters:

DN Attribute Name
The attribute name to use for an LDIF "dn" line.

Version Number
Displays a version attribute in the beginning of the output (required by RFC2849) if checked. This parameter is **On** by default.

Note: LDIF parser can suppress the LDIF version number by using the **Version Number** parameter.

Binary Attributes
If you need to specify additional attributes to be treated as binary (a binary attribute is returned as a byte array, not a string), specify them in this parameter. By default, the following attributes are treated as binary:

- photo
- personalSignature
- audio
- jpegPhoto
- javaSerializedData
- thumbnailPhoto
- thumbnailLogo
- userPassword
- userCertificate
- authorityRevocationList
- certificateRevocationList
- crossCertificatePair
- x500UniqueIdentifier
- objectGUID
- objectSid

Character Encoding
Character Encoding to be used; the default is UTF-8. Also see "Character Encoding conversion" on page 295.

Only Descriptive Records
If set, only write descriptive records. This parameter is **Off** by default.

An LDIF file may contain "change records" or "descriptive records". A change record describes some change that is needed for an entry. A descriptive record just describes an entry.

An easy way to see if a record is a change record, is that it will contain a "changetype" line as the second line, immediately after the "dn" line.

A correct LDIF File will either contain only change records, or only descriptive records.

The LDIF Parser uses the operation code of the work entry to decide if it should write a change record or just a descriptive record. That is, if the work Entry has any operation that is not generic, it assumes that it should write a change record. This is quite convenient, but it may not be what is wanted in all cases.

Even if the work Entry comes from a connector with Delta Enabled, it may be that the LDIF File should contain only descriptive records, for example because that is what can be read by the system that will use the LDIF File.

If this flag is set, only descriptive records will be generated, no matter what the operation of the Entry might be. If the operation is Delete, nothing will be written about that Entry, but otherwise the attribute values the Entry contains will be written as a descriptive record.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

See also

http://www.ietf.org/rfc/rfc2849.txt

Line Reader Parser

The Line Reader Parser reads single lines of data. The line read is returned in a single attribute. There is also an attribute named **linenumber** that contains the line number, starting with **1**.

Note: Use the Line Reader Parser if you want to copy a text file only. If you want to copy a binary file, see the FTP Object "Example" on page 526 for an example of how to copy a binary file.

The Line Reader Parser is useful when reading text files only.

Configuration

The Parser has the following parameters:

Attribute Name
Specifies the name of the attribute that contains the line of text either just read, or about to be written. Default is **line**.

Character Encoding
Character Encoding to be used; the default is UTF-8. Also see "Character Encoding conversion" on page 295.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Script Parser

The Script Parser enables you to write your own Parser using JavaScript.

To operate, a Script Parser must implement a few functions. The functions do not use parameters.

Note: The script for the Script Parser is running in a separate JavaScript Engine. This means that the script cannot access any variables that are available, or have been set, in the normal Hooks of an AssemblyLine.

Passing data between the hosting Connector and the script is done by using predefined objects. One of these predefined objects is the **result** object which is used to communicate status information. Upon entry into either function, the status field is set to **normal** which causes the hosting Parser to continue calls. Signaling end-of-input or errors is done by setting the status and message fields in this object. The **entry** object is populated on calls to **writeEntry** and is expected to be populated in the **readEntry** function. When reading entries you have the **inp** BufferedReader object available for reading character data from a stream. When writing entries you have the **out** BufferedWriter object available for writing character data to a stream.

You can add your own parameters to the configuration and obtain these by using the **parser** object.

Objects

Common objects (these are the same as for an AssemblyLine):

main The Config Instance (RS object) that is running.

task The AssemblyLine this Parser is a part of.

system
A `UserFunctions` object.

config The configuration for this element, that is, this Parser.

The following objects are the only ones accessible to the script Parser:

The result object

setStatus
code

- **0** - End of Input
- **1** - Status OK
- **2** - Error

setMessage
text

The entry object

addAttributeValue (name, value)
Adds a value to an attribute.

getAttribute (name)
Returns the named attribute.

A complete list of available methods, including parameters and return values, can be found in the Javadocs (*TDI_install_dir*/docs/api/com/ibm/di).

The inp object

read() Returns next character from stream.

readLine()
Returns next CRLF-stopped line from the input stream.

The out object

write (str)
Writes a string to the output stream.

writeln (str)
Writes a string followed by CRLF to the output stream.

The parser object

getParam(str)
Returns the parameter value associated with parameter name **str**

setParam(str, value)
Sets the parameter **str** to value **value**

logmsg(str)
Writes the parameter **str** in the log file

A complete list of methods can be found in the installation package.

The connector object

For more information, see the JavaDocs material included in the installation package.

Functions (methods)

The Parser should supply the following functions, where relevant for the intended usage in IBM Tivoli Directory Integrator:

readEntry()
Read the next logical entry from the input stream and populate the **entry** object. This function is not required for Parsers called in add_only situations only.

writeEntry()
Write the contents of the **entry** object to the output stream. This function is not required for Parsers that are only used for reading.

closeParser ( )
The closeParser function, if implemented, will be called when Connector.close is called. For example:

```
function closeParser ( )
{
  task.logmsg("CLOSE CALLED.");
}
```

flush()
The flush function will be called if the Parser's flush is called via the connector.getParser( ).flush( ) method. Implementing these methods in effect overrides the Parser's methods. For example:

```
function flush ( )
{
  task.logmsg("FLUSH CALLED.");
}
```

querySchema()
The querySchema function is called by the Parser's parent, that is the Connector into which this Parser has been configured. It is used to discover the Schema of the underlying data source, to populate the Input and Output maps. See “Schema” on page 333 for more information.

Configuration

The Parser has the following parameters:

External Files
If you want to include external script files at runtime, specify them here, one file on each line. These files are run before your script.

Include Global Scripts
Include scripts from the Script Library.

Character Encoding
Character Encoding to be used; the default is UTF-8. Also see "Character Encoding conversion" on page 295.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Script The user-defined script to run.

Note: When you use a Script Connector or Parser, the script is copied from the Library where it resides and into your configuration file. This has the advantage that you can customize the script, but with the disadvantage that new versions are not known to your AssemblyLine.

To work around this disadvantage, remove the old Script Parser from the AssemblyLine and re-introduce it.

Schema

A sample *querySchema* function is provided in the configuration parameter **Script**. It assumes the default case where we read from a text file one line at a time; hence, the schema returned from this function has only one field named 'line' of type java.lang.String. If you want specific behavior, you must override this function.

In it there are two predefined objects which are accessible from these two script objects:

list This is a Vector object. The `querySchema(Object)` function should add Entry objects to this Vector.

Source
This is an Object parameter passed to the `querySchema(Object)` function when it is called.

For building a meaningful query schema you must populate the predefined **list** object with entries containing at least one attribute called "name" and optional attributes: "syntax" or "extsyntax". This can be done by creating an *Entry* object and calling its addAttributeValue function to set the desired values to the attributes.

According to the success in retrieving schema you can set three different types of exit codes, calling the predefined result object's function setStatus(int) with one of the following values:

- 0 - End of Input
- 1 - Status OK
- 2 - Error

For setting more detailed information about the result you can use the setMessage(String) function which takes one textual parameter. Only if the exit code is 1 a schema is returned by the querySchema(Object) function, otherwise null is returned.

Example

Go to the *root_directory*/examples/script_parser directory of your IBM Tivoli Directory Integrator installation.

See also

“Script Connector” on page 255,
“Scripted Function Component” on page 393
"JavaScript Parser" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.

Simple Parser

The Simple Parser reads and writes entries. The format is lines with *attributename*:*value* pairs, where *attributename* is the name of the attribute, and *value* is the value.

Multi-valued attributes use multiple lines. Lines with a single period mark the end of an entry. **\r** and **\n** in the *value* is an encoding of CR and LF.

Configuration

The Parser has the following parameters:

Character Encoding
: Character Encoding to be used; the default is UTF-8. Also see "Character Encoding conversion" on page 295.

Detailed Log
: If this parameter is checked, more detailed log messages are generated.

SOAP Parser

The SOAP Parser reads and writes SOAP XML documents. The Parser converts SOAP XML documents to or from entry objects in a simple, straightforward fashion. When writing the XML document, the Parser uses attributes from the entry to build the document. The **SOAP_CALL** attribute is expected to contain the value for the SOAP call. Similarly, when reading, this attribute is set to reflect the first tag following the **SOAP-ENV:Body** tag. Then, for each attribute in the entry, a tag with that name and value is created. When reading the document, each tag under the **SOAP_CALL** tag translates to an attribute in the entry object.

Note: When working with the WebServices Connector, you must avoid starting attribute names with special characters (such as **[0-9] [- ' ( ) + , . / = ? ; ! * # @ $ %]**). Also, you must avoid having attribute names that include special characters (such as **[' ( ) + , / = ?; ! * # @ $ %]**). This is because WebServices builds on SOAP, which is XML. XML does not accept $ as in tags.

The following examples show an entry and a SOAP XML document as they are read or written.

Example Entry

```
*** Begin Entry Dump
 SOAP_CALL: 'updateLDAP'
 mail: ('john@doe.com)'
 uid: 'johnd'
*** End Entry Dump
```

Example SOAP document

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="(http://schemas.xmlsoap.org/soap/envelope/)"
 xmlns:xsi="(http://www.w3.org/1999/XMLSchema-instance)"
 xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:updateLDAP xmlns:ns1="" SOAP-ENV:encodingStyle=
   "http://schemas.xmlsoap.org/soap/encoding/">
<uid xsi:type="xsd:string">johnd</uid>
<mail xsi:type="xsd:string">john@doe.com</mail>
</ns1:updateLDAP>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Configuration

The Parser has the following parameters:

Omit XML Declaration
Omit the XML declaration header in the output stream.

Document Validation
Request a DTD/XSchema-validating XML parser.

Namespace Aware
Request a namespace-aware XML parser.

Character Encoding
Character Encoding to be used; the default is UTF-8. Also see “Character Encoding conversion” on page 295.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Parser-specific calls

You can access the SOAP Parser from your script by dynamically loading the Parser and calling the methods to read or write SOAP documents. The following example shows how to generate the XML document from an entry:

```
var e = system.newEntry();
e.setAttribute ("soap_call", "updateLDAP");
e.setAttribute ("uid", "johnd");
e.setAttribute ("mail", "(john@doe.com)");

// Retrieve the XML document as a string
var soap = system.getParser ("ibmdi.SOAP");
soap.initParser();
var soapxml = soap.getXML ( e );

task.logmsg ( "SOAP XML Document" );
task.logmsg ( soapxml );

// Write to a file
var soap = system.getParser("ibmdi.SOAP");
soap.setOutputStream ( new java.io.FileOutputStream("mysoap.xml") );
soap.writeEntry ( e );
soap.close();

// Read from file
soap.setInputStream ( new java.io.FileInputStream ("mysoap.xml") );
var entry = soap.readEntry();

// Read from string (from soapxml generated above)
var entry = soap.parseRequest( soapxml );
task.dumpEntry ( entry );
```

Examples

Go to the *root_directory*/examples/soap directory of your IBM Tivoli Directory Integrator installation.

SPMLv2 Parser

Introduction

SPML Version 2 (SPMLv2) defines a core protocol [SPMLv2] over which different data models can be used to define the actual provisioning data. The combination of a data model with the SPML core specification is referred to as a profile. The use of SPML requires that a specific profile is used, although the choice of which profile is used to negotiated out-of-band by the participating parties.

The DSML v2 protocol [DSMLV2] was designed to perform LDAP type operations using web services. The DSML V2 protocol defines synchronous request/response semantics and a data model based on attribute/value pairs. DSML V2 does not define an attribute/value pairs schema mechanism.

The SPMLv2 Parser supports the SPMLv2 DSMLv2 Profile. It is a Tivoli Directory Integrator Parser component that parses and creates SPMLv2 messages, that is, it is intended to parse individual SPMLv2 requests and responses or write SPMLv2 requests and responses.

The SPMLv2 Parser supports core operations as specified in the "(SPML) v2 - DSML v2 Profile" specification. Explicit Tivoli Directory Integrator Entry schemas are defined for each of the supported operations.

The Parser extends the “XML Parser” on page 351 and has the ability to read enormous requests/responses without creating all the SPML messages in the memory. In addition, it has been implemented on top of the OpenSPML 2.0 Toolkit.

The Parser is capable of reading/writing Batch messages. The following types from the toolkit have been used:

```
org.openspml.v2.msg.spmlbatch.BatchRequest;
org.openspml.v2.msg.spmlbatch.BatchResponse;
```

On each **readEntry** call the Parser will return an Entry representing the individual request(s) or response(s) contained in the batch message. On **writeEntry** the Parser will write individual request(s) or response(s) inside the appropriate batch message.

Note: This component is not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.

Operations

A conformant provider must implement all the operations defined in the Core XSD. The following are the core operations:

- Add (Add Request and Add Response)
- Modify (Modify Request and Modify Response)
- Delete (Delete Request and Delete Response)
- Lookup (Lookup Request and Lookup Response)

The Parser also supports Search operations:

- Search (Search Request and Search Response)

Add request

Entries with the following structure are parsed (on read) and created (on write) by the Parser for Add Requests:

Table 55.

Attribute	Value
spml.operation	set to Add

Table 55. (continued)

Attribute	Value
spml.operation.type	set to Request
spml.containerID	set to the ID attribute's value of a containerID element if it is present as a subelement of the addRequest element.
spml.containerID.targetID	set to the ID attribute's value if a targetID attribute is present in the containerID element.
spml.requestID	a reasonably unique value that identifies each outstanding request.

Additionally, for each DSML attr element: an attribute named as the "name" XML attribute of the DSML "attr" element and as value(s) specified for the "attr" DSML element.

Add response

Entries with the following structure are parsed (on read) and created (on write) by the Parser for Add Response:

Table 56.

Attribute	Value
spml.operation	set to Add
spml.operation.type	set to Response
spml.psoID	set to the ID attribute's value of the "psoID" element if a "psoID" element is available in the response.
spml.pso.targetID	set to the targetID attribute's value of the psoID element if the provider supports more than one target.
spml.requestID	reasonably unique value that identifies each outstanding request.
spml.errorCode	created if the add request has failed. The value of this must characterize the failure.
spml.status	holds the status attribute's value of the AddResponse element.
spml.errorMessages	an array of string objects that provides additional information about the status or failure of the requested operation.

Modify request

One important thing to mention is that the modify operation may change the identifier of the modified object.

Entries with the following structure are parsed (on read) and created (on write) by the Parser for Modify Requests:

Table 57.

Attribute	Value
spml.operation	set to Modify
spml.operation.type	set to Request
spml.psoID	set to the ID attribute's value. The Modify Request must always contain a <psoID> element that identifies an object that exists on a target, exposed by the provider.
spml.pso.targetID	this attribute may not be specified if the provider supports only one target.
spml.requestID	a reasonably unique value that identifies each outstanding request.

In addition, for each modification item: an attribute named as the "name" XML attribute of the DSML "modification" element, with the values specified for the "modification" DSML element and Tivoli Directory Integrator attribute's operation set as the "operation" XML attribute of the DSML "modification" element.

Modify response

Entries with the following structure are parsed (on read) and created (on write) by the Parser for Modify Responses:

Table 58.

Attribute	Value
spml.operation	set to Modify
spml.operation.type	set to Response
spml.psoID	If the provider successfully modified the requested object, the <modifyResponse> must contain a <pso> element. The <pso> contains the subset of (the XML representation of) a requested object that the "returnData" attribute of the <modifyRequest> specified.
spml.pso.targetID	this attribute may not be specified if the provider supports only one target.
spml.status	the status attribute's value of the ModifyResponse element
spml.errorCode	created if the request has failed. The value of this must characterize the failure. This attribute may have one of predefined values by the SPML specification.
spml.errorMessages	an array of string objects that provides additional information about the status or failure of the requested operation. This attribute is optional.
spml.requestID	reasonably unique value that identifies each outstanding request.

Delete request

Entries with the following structure are parsed (on read) and created (on write) by the Parser for Delete Requests:

Table 59.

Attribute	Value
spml.operation	set to Delete
spml.operation.type	set to Request
spml.psoID	set to the ID attribute' value of the <psoID> element. The Delete Request must always contain the PSO Identifier.
spml.pso.targetID	attribute may not be specified if the provider supports only one target.
spml.requestID	a reasonably unique value that identifies each outstanding request.

Delete response

Entries with the following structure are parsed (on read) and created (on write) by the Parser for Delete Responses:

Table 60.

Attribute	Value
spml.operation	set to Delete
spml.operation.type	set to Response
spml.errorCode	created if the request has failed. The value of this must characterize the failure. This attribute may have one of predefined values by the SPML specification.
spml.status	holds the status attribute's value of the DeleteResponse element.

Table 60. (continued)

Attribute	Value
spml.errorMessages	an array of string objects that provides additional information about the status or failure of the requested operation. This attribute is optional.
spml.requestID	a reasonably unique value that identifies each outstanding request.

Lookup request

Entries with the following structure are parsed (on read) and created (on write) by the Parser for Lookup Requests:

Table 61.

Attribute	Value
spml.operation	set to Lookup
spml.operation.type	set to Request
spml.psoID	set to the ID attribute's value of the <psoID> element. The Lookup Request must always specify PSO identifier.
spml.pso.targetID	attribute may not be specified if the provider supports only one target.
spml.requestID	a reasonably unique value that identifies each outstanding request.

Lookup response

Entries with the following structure are parsed (on read) and created (on write) by the Parser for Lookup Response:

Table 62.

Attribute	Value
spml.operation	set to Lookup
spml.operation.type	set to Response
spml.psoID	set to the ID attribute's value of the <psoID> element.
spml.pso.targetID	equal to the "targetID" attribute of the <psoID> element. It may be not specified if the provider supports only one target.
spml.status	holds the status attribute's value of the LookupResponse element
spml.requestID	a reasonably unique value for the "requestID" attribute in each request. A "requestID" value need not be globally unique. A "requestID" needs only be sufficiently unique to identify each outstanding request.
spml.errorMessage	an array of string objects that provides additional information about the status or failure of the requested operation. This attribute is optional.

Search request

Entries with the following structure are parsed (on read) and created (on write) by the Parser for Search Requests:

Table 63.

Attribute	Value
spml.operation	set to Search
spml.operation.type	set to Request
spml.scope	search scope
spml.containerID	contains value of the ID attribute of the "basePsoID" element of the Search Request

Table 63. (continued)

Attribute	Value
spml.containerID.targetID	contains value of the targetID attribute of the "basePsoID" element of the Search Request
spml.attributeDescription	multi-valued Attribute whose String values hold the names of the attributes listed in the "attributes" element of the Search Request.
spml.filter.substrings.name	contains the value of the Name of the Filter Substrings element
spml.filter.substrings.initial	the value of the Initial element of the Filter Substrings element
spml.filter.substrings.any	multi-valued Attribute which contains the values of the Any element of the Filter Substrings element
spml.filter.substrings.final	contains the value of the Final of the Filter Substrings element
spml.filter	contains the value of the Filter element as a hierarchical Attribute object; this uses v7.0 hierarchical objects

See "Search Filter capabilities" on page 344 for a more thorough discussion of search filters.

Search response

Entries with the following structure are parsed (on read) and created (on write) by the Parser for Search Responses:

Table 64.

Attribute	Value
spml.operation	set to Search
spml.operation.type	set to Response
spml.resultEntries	a multi valued attribute, each of its values is an Entry whose attributes correspond to the "<spml:data>\attr" elements of the individual "<spml:pso>" element.
spml.errorCode	created if the request has failed. The value of this must characterize the failure. This attribute may have one of predefined values by the SPML specification.
spml.status	holds the status attribute's value of the SearchResponse element.
spml.errorMessages	an array of string objects that provides additional information about the status or failure of the requested operation. This attribute is optional.
spml.requestID	a reasonably unique value that identifies each outstanding request.

Binary and non-String Attributes

When parsing SPML messages, attributes tagged as binary by the **Binary Attributes** Parser parameter are Base64 decoded, that is, the string value from the SPML message is Base64 decoded to Java byte array.

When creating SPML messages, all Attributes whose value is Java byte array are Base64 encoded to String before being written in the SPML message.

If when creating a SPML message an Attribute is passed whose value's type is neither String, nor Java byte array, the value is converted to String by calling the object's `toString()` method and this String value is stored in the SPML message.

Attribute operation tagging

The SPMLv2 Parser handles the different modification operations by tagging the Entry attributes according to the *dsml:modification* operation attribute value in the SPMLv2 document.

For example, when reading the following SPML structure:

```
<modifyRequest xmlns='urn:oasis:names:tc:SPML:2:0' returnData='everything'>
 <psoID ID='CN=DnsUpdateProxy,OU=Groups,DC=2k3,DC=dom'/>
 <modification>
  <dsml:modification xmlns:dsml='urn:oasis:names:tc:DSML:2:0:core' name='member' operation='delete'>
   <dsml:value>CN=Eric Clapton,CN=Users,DC=2k3,DC=dom</dsml:value>
  </dsml:modification>
  <dsml:modification xmlns:dsml='urn:oasis:names:tc:DSML:2:0:core' name='member2' operation='add'>
   <dsml:value>CN=Eric Adams,CN=Users,DC=2k3,DC=dom</dsml:value>
  </dsml:modification>
  <dsml:modification xmlns:dsml='urn:oasis:names:tc:DSML:2:0:core' name='member3' operation='replace'>
   <dsml:value>CN=Joey DeMaio,CN=Users,DC=2k3,DC=dom</dsml:value>
  </dsml:modification>
 </modification>
</modifyRequest>
```

If you parse this SPML extract into an Entry you will have to map three Attributes named "member", "member2 " and "member3"; each one will be tagged with the according operation (that is, delete, add, replace). The value of the modification operation can also be accessed via script:

```
work.getAttribute("member").getOperation();
```

When parsing an Entry to a SPML document the attribute will be tagged with the attribute operation provided in the work Entry. If this modification operation is not set, the default is used - "replace". Alternatively you can set it manually via script:

```
work.getAttribute("member").setOperation("add");
```

Search Filter capabilities

In earlier versions of Tivoli Directory Integrator, the SPMLv2 Parser supported the Substrings Filter element only in Search requests and used the following attributes: spml.filter.substrings.name, spml.filter.substrings.initial, spml.filter.substrings.final and spml.filter.substrings.any to contain the Substrings subelements values.

The current version of the SPMLv2 Parser can parse the other filtering features provided by DSMLv2:

- **not** - negation of contained filter item;
- **and** - logical 'and' containing several filter items
- **or** - logical 'or' containing several filter items
- **equalityMatch** - filter item indicating equal match
- **approxMatch** - filter item indicating approximate match
- **extensibleMatch** - filter item indicating extensible match
- **greaterOrEqual** - filter item indicating match if greater or equal
- **lessOrEqual** - filter item indicating match if less or equal
- **present** - filter item indicating presence of specified attribute

Reading: When reading a Filter element, the parser creates a corresponding hierarchical Attribute named "spml.filter". For every filter item a separate Attribute object is created and appended as a child; correspondingly, for every subelement or another filter item contained in a filter item separate Attributes are created and added as its children.

Note: According to the DSMLv2 schema the **not** element can contain only one filter item.

The following attributes: spml.filter.substrings.name, spml.filter.substrings.initial, spml.filter.substrings.final and spml.filter.substrings.any are created only when the Filter element contains a single Substrings element; otherwise a spml.filter hierarchical attribute is created.

Limitation: When reading the SPMLv2 Parser cannot read the matchingRule and dnAttributes (always has default value of false) attributes of the extensibleMatch element. This is caused by the underlying

Open SPML 2.0 library that tries to read the matchingRule and dnAttributes as values instead of attributes. However when writing these attributes are written correctly.

Writing: When writing, if a smpl.filter attribute is present in the provided entry it is used and spml.filter.substrings.name, spml.filter.substrings.initial, spml.filter.substrings.final and spml.filter.substrings.any attributes are ignored. If a smpl.filter attribute is not present the spml.filter.* attributes are used (backward compatibility).

Here is an example hierarchical spml.filter attribute and the corresponding XML generated from it:

Table 65. Hierarchical spml.filter attribute

spml.filter attribute	Filter element

'"and": {
"or": {
"substrings": {
"name": "cn",
"initial": "J"
},
"not": {
"and": {
"lessOrEqual": {
"name": "roomnumber",
"value": "2000"
},
"greaterOrEqual": {
"name": "roomnumber",
"value": "3000"
}
}
},
"approxMatch": {
"name": "sn",
"value": "Smith"
}
},
"equalityMatch": {
"name": "objectClass",
"value": "inetorgperson"
}
}'
	<dsml:filter xmlns:dsml='urn:oasis:names:tc:DSML:2:0:core'>
	<dsml:and>
	<dsml:or>
	<dsml:substrings name='cn'>
	<dsml:initial>J</dsml:initial>
	</dsml:substrings>
	<dsml:not>
	<dsml:and>
	<dsml:lessOrEqual name='roomnumber'>
	<dsml:value>2000</dsml:value>
	</dsml:lessOrEqual>
	<dsml:greaterOrEqual name='roomnumber'>
	<dsml:value>3000</dsml:value>
	</dsml:greaterOrEqual>
	</dsml:and>
	</dsml:not>
	<dsml:approxMatch name='sn'>
	<dsml:value>Smith</dsml:value>
	</dsml:approxMatch>
	</dsml:or>
	<dsml:equalityMatch name='objectClass'>
	<dsml:value>inetorgperson</dsml:value>
	</dsml:equalityMatch>
	</dsml:and>
	</dsml:filter>

Configuration

The Parser needs the following configuration parameters:

Binary Attributes
Specifies a comma delimited list of attributes that will be treated by the Parser as binary attributes (Base64 decoded/encoded as required).

This parameter has the following default list of attributes that you can change: photo, personalSignature, audio, jpegPhoto, javaSerializedData, thumbnailPhoto, thumbnailLogo, userPassword, userCertificate, authorityRevocationList, certificateRevocationList, crossCertificatePair, x500UniqueIdentifier, objectGUID, objectSid.

Character Encoding
Character encoding to use when reading or writing. The default is UTF-8. Since this parser extends the XML Parser, the same considerations as for that Parser apply.

Detailed Log
Checking this item will cause detailed logs to be generated.

Example

Examples of how to use this Parser have been provided in the *TDI_install_dir*/examples/SPMLv2Parser directory.

See also

"DSMLv2 Parser" on page 305

Simple XML Parser

The Simple XML Parser reads and writes XML documents; it deals with XML data which is not more than two levels deep. This Parser uses the Apache Xerces and Xalan libraries. The Parser gives access to XML document through a script object called **xmldom**. The **xmldom** object is an instance of the org.w3c.dom.Document interface. Refer to http://java.sun.com/xml/jaxp-1.0.1/docs/api/index.html for a complete description of this interface.

You can also use the XPathAPI (http://xml.apache.org/xalan-j/apidocs/index.html and access its Java Classes in your Scripts) to search and select nodes from the XML document. **selectNodeList**, a convenience method in the **system** object, can be used to select a subset from the XML document.

When the Connector is initialized, the Simple XML Parser tries to perform Document Type Definition (DTD) verification if a **DTD** tag is present.

Use the Connector's override functions to interpret or generate the XML document yourself. Create the necessary script in either the **Override GetNext** or **GetNext Successful** in your AssemblyLine's hook definitions. If you do not override, the Parser reads or writes a very simple XML document that mimics the entry object model. The default Parser only permits you to read or write XML files two levels deep. It will also read multi-valued attributes, although only one of the multi-value attributes will be shown when browsing the data in the Schema tab.

Note that certain methods, such as *setAttribute* are available in both the IBM Tivoli Directory Integrator **entry** and the objects returned by **xmldom.createElement**. These functions have the same name or signature. Do not confuse the **xmldom** objects with the IBM Tivoli Directory Integrator objects.

Notes:

1. This Parser was called "XML Parser" in pre-Tivoli Directory Integrator 7.0 releases. In Tivoli Directory Integrator 7.0 it is renamed to Simple XML Parser and a new XML Parser was added; see “XML Parser” on page 351. The new Parser has a lot of improvements and is now the main Tivoli Directory Integrator XML Parser.
2. If you read large (more than 4MB) or write large (more than 14MB) XML files, your Java VM may run out of memory. Refer to "Increasing the memory available to the Virtual Machine" in *IBM Tivoli Directory Integrator V7.1 Users Guide* for a solution to this. Alternatively, use the “XML Parser” on page 351 or the “XML SAX Parser” on page 361.
3. The Parser silently ignores empty entries.
4. When reading a CDATA attribute, no blank space is trimmed from the value. However, blank space is trimmed from attributes that are not CDATA.
5. Certain characters, such as **$**, are illegal in XML tags. Avoid these characters in your attribute names when using the XML Parser because these characters might create illegal XML.
6. When reading from an LDAP directory or an LDIF file, the distinguished name (DN) is typically returned in an attribute named **$dn**. If you map this attribute without changing the name into an XML file, it fails because $dn is not a legal tag in an XML document. If you do explicit mapping, you must change "$dn" to "dn" (or something without a special character) in your output Connector. If you do implicit mapping, for example, * or **Automatically map all attributes** checked in the **AssemblyLine Settings** (through the **Config . . .** tab of the AssemblyLine), you can configure the XML Parser to translate the distinguished name (for example, $dn) to a different name. For example, you can add something like this in the **Before GetNext** Hook:

```
conn.setAttribute("dn", work.getAttribute("$dn"));
conn.removeAttribute("$dn");
```

Configuration

The Parser has the following parameters:

Root Tag
The root tag (output).

Entry Tag
The entry tag for entries (output).

Value Tag
The value tag for entry attributes (output).

Character Encoding
Character Encoding to be used. See "Character Encoding in the Simple XML Parser."

Omit XML Declaration
If checked, the XML declaration is omitted in the output stream.

Document Validation
If checked, this parser requests a DTD/Schema-validating parser.

Namespace Aware
If checked, this parser requests a namespace-aware parser.

Indent Output
If this field is checked, then the output is indented.

Note: If this text is to be processed by a program (and not meant for human interpretation) you most likely will want to deselect this parameter. This way, no unnecessary spaces or newlines will be inserted in the output.

Detailed Log
If this parameter is checked, more detailed log messages are generated.

Character Encoding in the Simple XML Parser

The default and recommended Character Encoding to use when deploying the Simple XML Parser is UTF-8. This will preserve data integrity of your XML data in most cases. When you are forced to use a different encoding, the Parser will handle the various encodings in the following way:

- When reading a file, parser will look for encoding in the following order:
  1. If the Tivoli Directory Integrator CharacterSet config parameter is set, the encoding is set to the value specified in this parameter. However, check #2 is attempted and will overwrite this check if successful when the encoding specified is UTF-32 or UTF-16.
  2. The XML is checked for the existence of an encoding attribute from the XML declaration. First, the XML is checked to see if a BOM exists. If it does, the encoding specified in the BOM is used to retrieve the encoding attribute from the XML declaration. Otherwise, the default encoding of the JRE is used to retrieve the attribute. If the encoding attribute from the XML declaration is found, this value will be used.
  3. If the Tivoli Directory Integrator CharacterSet was not set and no encoding attribute from the XML declaration is found, then the BOM encoding will be used if it is set.
  4. The default encoding of the JRE is used if none of the above are true.
- On output, the Parser will write an XML header specifying the character encoding. This will be the encoding specified in the Parser config. If nothing is specified there, UTF-8 will be used.

Examples

Override Add hook:

```
var root = xmldom.getDocumentElement();
var entry = xmldom.createElement ("entry");
var names = work.getAttributeNames();

for ( i = 0; i < names.length; i++ ) {
 xmlNode = xmldom.createElement ("attribute");
```

```
 xmlNode.setAttribute ( "name", names[i] );
 xmlNode.appendChild ( xmldom.createTextNode ( work.getString(
   names[i] ) ) );
 entry.appendChild ( xmlNode );
}
root.appendChild ( entry );
```

After Selection hook:

```
//
// Set up variables for "override getnext" hook
//

var root = xmldom.getDocumentElement();
var list = system.selectNodeList ( root, "//Entry" );
var counter = 0;
```

Override GetNext hook

```
//
// Note that the Iterator hooks are NOT called when we override the
  getnext function
// Initialization done in After Select Entries hook


var nxt = list.item ( counter );

if ( nxt != null ) {
   var ch = nxt.getFirstChild();
   while ( ch != null ) {
      var child = ch.getFirstChild();
      while (child != null ) {
        // Use the grandchild's value if it exist, to be able to
    read multivalue attributes
 grandchild = child.getFirstChild();
 if (grandchild != null)
   nodeValue = grandchild.getNodeValue();
      else nodeValue = child.getNodeValue();
 // Ignore strings containing newlines, they are just fillers
      if (nodeValue != null && nodeValue.indexOf('\n')
    == -1) {
            work.addAttributeValue ( ch.getNodeName(), nodeValue );
      }
 child = child.getNextSibling();
      }
      ch = ch.getNextSibling();
   }

   result.setStatus (1); // Not end of input yet
   counter++;
} else {
   result.setStatus (0); // Signal end of input
}
```

The previous example parses files containing items that look like the following entries:

```
<DocRoot>
  <Entry>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
    <title>Engineer</title>
  </Entry>
  <Entry>
    <firstName>Al</firstName>
    <lastName">Bundy</lastName>
    <title">Shoe salesman</title>
  </Entry>
</DocRoot>
```

Suppose instead that the input looks like the following entries:

```
<DocRoot>
  <Entry>
    <field name="firstName">John</field>
    <field name="lastName">Doe</field>
    <field name="title">Engineer</field>
  </Entry>
  <Entry>
    <field name="firstName">Al</field>
    <field name="lastName">Bundy</field>
    <field name="title">Shoe salesman</field>
  </Entry>
</DocRoot>
```

Here the attribute names can be retrieved from attributes of the field node, and this code is used in the **Override GetNext** Hook:

```
var nxt = list.item ( counter );

if ( nxt != null ) {
 var ch = nxt.getFirstChild();
 while ( ch != null ) {
  if(String(ch.getNodeName()) == "field") {
   attrName = ch.getAttributes().item(0).getNodeValue();
   nodeValue = ch.getFirstChild().getNodeValue();
   work.addAttributeValue ( attrName, nodeValue );
  }
  ch = ch.getNextSibling();
 }

 result.setStatus (1); // Not end of input yet
 counter++;
} else {
 result.setStatus (0); // Signal end of input
}
```

This example package demonstrates how the base Simple XML Parser functionality can be extended to read XML more than two levels deep, by using the **Override GetNext** and **Override Add** hooks.

Additional Examples

Go to the *root_directory*/examples/simplexmlparser directory of your IBM Tivoli Directory Integrator.

See also

"XML Parser" on page 351,
"XML SAX Parser" on page 361,
"XSL based XML Parser" on page 365,
"SOAP Parser" on page 337,
"DSML Parser" on page 303.

XML Parser

This XML Parser is introduced for the first time in Tivoli Directory Integrator v7.0. It uses the XLXP implementation of the StAX (JSR-173) specification. StAX is a cursor based XML parser, capable of both reading and writing XML.

Note: The traditional DOM-based Parser available in older versions of Tivoli Directory Integrator has been renamed, and is now available as the "Simple XML Parser" on page 347. The new XML Parser is deemed a replacement for the older component, and you are encouraged to migrate your older Configs to use the new Parser.

Introduction

A Connector uses the XML Parser to either retrieve a Tivoli Directory Integrator Entry object from source XML or output a Tivoli Directory Integrator Entry object as XML. The XML Parser uses the StAX cursor based parser internally. In previous versions of the Tivoli Directory Integrator XML Parser (now the Simple XML Parser) the DOM mechanism was used for parsing a XML. The main advantages of the StAX implementation is that now the Tivoli Directory Integrator Parser is much faster because it does not need to load the whole XML structure in memory like DOM does. Because of its memory efficiency the StAX implementation is more suited when the Tivoli Directory Integrator solution is supposed to deal with unusually large XML structures.

The only drawback of this memory efficient mechanism of parsing an XML data is that no random element access is available since all StAX does is running through an XML structure and pulls one element at a time. Depending on the configuration of the Tivoli Directory Integrator XML Parser each one of the elements pulled out could be either skipped or put in an Entry with Attributes representing each element being pulled out of the XML.

Configuration

The XML Parser has the following configurable parameters:

Simple XPath
Contains the value used (an XPath-like expression) to discover elements to interpret them as entries. This parameter is also used to display the structure of the XML document to be written.

Entry Tag
Holds the name of the element that will wrap each entry passed to the XML Parser.

Value Tag
Holds the name of the element that will wrap each attribute value passed to the XML Parser.

Prefix to Namespace Map
Mappings between <prefix>=<namespace> separated by the pipe char (|). If the prefix starts with $ it will be considered as a default namespace declaration. The default value is "<prefix>=<namespace>".

XSD Schema Location
The schema location, used for display purposes only.

Character Encoding
Character encoding to use when reading or writing. The default is UTF-8; also see "Character Encoding in the XML Parser" on page 357.

Static Attributes Declaration
Used to declare attributes and prefixes. They will be written with the static elements read form the **Simple XPath** parameter. This is a text area, and the default is:

```
<!-- this is an example for statically declared XML attributes/namespaces -->
<!-- DocRoot xmlns="defaultNS" attr1="val2">
 <Entry xmlns:p1="p1NS" p1:attr2="val2" />
</DocRoot-->
```

Ignore repeating XML declarations while reading
Check this to always acknowledge the first XML declaration (if any), any subsequent other ones will be ignored. The default value is unchecked.

Coalescing
If checked, then the Parser will coalesce adjacent character data sections. The default value is unchecked.

Omit XML declarations when writing
Check this to suppress writing an XML declaration to the output. Useful for appending to an existing XML file. The default value is unchecked.

Multi-rooted Document
If checked, output each Entry as a standalone element. This will create a multi-rooted document. The default value is unchecked.

Indent Output
If this field is checked, then the XML output is indented. The default value is checked.

Detailed log
Check this to generate more detailed log messages.

Using the Parser

Navigation through the XML structure

The XML Parser recognizes very simple XPath expressions. According to the expression the parser finds and returns an Entry that will either contain a single Attribute object representing the element itself or multiple Attribute objects in case the wrapping/unwrapping function of the parser is utilized. Current XPath implementations require random element access (over an Object Model) to pinpoint the element(s) referred by the XPath expression. Since a StAX parser does not provide this feature (random element access) it can only work with simple XPath expressions like these:

- /root/container1/container2/entry
- /root/prefix:container/entry
- /root/$prefix:container/
- /root/*/entry
- /root/prefix:*
- /root/$prefix:*/entry

Navigation when reading: You can provide several simple paths if the structure of the XML is quite complex. Each XPath expression is separated from the previous using the pipe char – "|". Each expression is used for finding elements in the XML document. By default the XML Parser is able to work with XMLs with two-levels in depth, just like the Simple XML Parser can. In additionally the XML Parser provides an easy way for working with arbitrary deep and complex hierarchical structures. For more details, take a look at these two sections:

Simple XML: This is the default way of parsing an XML. Just like the Simple XML Parser this parser is able to parse a XML structure like this one:

```
<?xml version="1.0" encoding="UTF-8" ?>
<DocRoot>
    <Entry>
        <telephoneNo>
                <ValueTag>555-888-8888</ValueTag>
                <ValueTag>555-999-9999</ValueTag>
        </telephoneNo>
        <User>Jill Vox</User>
    </Entry>
</DocRoot>
```

When in simple mode the XML parser will make sure that some of the elements in the hierarchy are stripped off to return a simple, flat-like data structure (Entry). The behavior of the parser is controlled by three parameters:

1. **Simple XPath** (xpath.expr) field – used to specify the path to the container element which will be searched for the presents of the element specified by the entry.tag parameter. By default this field is configured to find the root element of the input XML.
2. **Entry Tag** (entry.tag) field – used to specify the name of the element that represents the entry that will be returned.

   Note: The presence of this parameter specifies whether the parser will do a simple or advanced parsing. If this parameter is empty the XML Parser will do advanced parsing.
3. **Value Tag** (value.tag) field – used to specify the name of the element that holds a value of a multi-valued attribute.

   Note: This parameter is not used if the **entry.tag** parameter is empty.

Note: The **xpath.expr** parameter can be used in conjunction with the **ns.map** parameter to filter some of the elements. For more details see the Advanced XML section.

Using the default values of these parameters the XML Parser can easily parse the example XML above and an entry with the following data will be returned:

```
{
 "telephoneNo": [
  "555-888-8888",
  "555-999-9999"
 ],
 "User": "Jill Vox"
}
```

Here the "Entry" element has been removed and also the ValueTag elements have been taken as values of the "telephoneNo" attribute.

Note: If the structure of the input XML is not known prior to reading then you can remove the value of the **entry.tag** parameter. This way the whole XML is read at once and will show you what the XML structure looks like. Based on the returned information you can then reconfigure the parser to match the XML structure.

Advanced XML: The XML Parser runs in this mode when the entry.tag parameter is empty.

For each element found only a single Attribute object will be created. On each cycle the XML Parser returns an Entry object which contains only one Attribute that corresponds to the element found in the XML document. The XML Parser returns null if no element that matches any of the XPath expressions is found in the XML document.

There are two parameters that configure the way the parser finds data in the XML.

1. **Simple XPath** (xpath.expr) parameter – used to specify the path to the element which contains the desired data. This parameter is required.
2. **Prefix To Namespace Map** (ns.map) field – used to declare prefixes and namespaces. As we will see this is not a required parameter but provides more flexibility for finding specific data.

In order to fully describe these two parameters consider this example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root xmlns="defaultNS" xmlns:pref1="prefix1NS">
 <pref1:container xmlns:pref2="prefix2NS" attribute1="attrValue1" pref1:attribute2="attrValue2">
  <pref2:entryElement>
   <someData />
```

```
  </pref2:entryElement>
  <pref2:entryElement xmlns:pref2="prefix3NS">
   <moreData />
  </pref2:entryElement>
 </pref1:container>
</root>
```

Let's say that the desired data we need to get is in any of the entry elements. The simplest way to get each entry element is to specify the following element:

```
xpath.expr: /root/container/entryElement
```

Each iteration will get a single entryElement. For this example we would need two iterations to get both of the entryElement elements. Without specifying the element's prefix or namespace the parser will match any element using the local name we give in the Simple XPath expression.

However you may notice that both entryElements are different since they belong to different namespaces. Let's say that the desired data is the entryElement that belongs to the "prefix3NS" namespace. Using the previous configuration will get us data that is not needed (i.e. the first entryElement). This is where the ns.map comes in since we need to tell the parser where the desired element belongs to. Here is how we get only the second element:

```
xpath.expr: /root/container/pref2:entryElement
ns.map: pref2=prefix3NS
```

Here the parser will match the element's local name (that is, entryElelement) and the namespace. If we do not specify the pref2 in the ns.map field, then the parser will use only the prefix and the local name found in the xpath.expr expression when it does the matching. If we redefine the pref2 in the ns.map the latest definition will be used and any previous will be ignored.

```
xpath.expr: /root/container/p1:entryElement | /root/container/p2:entryElement
ns.map: p1=prefix2NS | p2=prefix3NS
```

In this case the prefixes are ignored and only the elements' local names and namespaces are considered.

The following expression:

```
xpath.expr: /$defPref:root/container/pref2:entryElement
ns.map: pref2=prefix3NS | $defPref= defaultNS
```

This has the same meaning as the second example configuration. However the expression $defPref tells the parser that the root element belongs to the default namespace "defaultNS". This is useful when the default namespace have been predefined in the XML at some place. In this example the parser will only match the local name and the namespace but will expect the XML element it checks belongs to the default namespace (that is, has no prefix). In other words this:

```
xpath.expr: /$defPref:root/$somePref:container/pref2:entryElement
ns.map: pref2=prefix3NS | $somePref=prefix1NS | $defPref= defaultNS
```

will not return any entryElement elements.

The XML Parser has the ability to navigate the XML tree using wildcards. The supported wildcard is the asterisk character – "*", which is used to replace the local name of an element of the XML. Let's say the following configuration is set:

```
xpath.expr: /root/container/*
```

This expression would retrieve each element under the element with local name "container", thus resulting in two iterations in total. The result would be the same if the xpath.expr is set to "/root/container/pref2:*" and the pref2 is not defined in the ns.map field.

The following configuration:

```
xpath.expr: /root/container/p1*
ns.map: p1=prefix2NS
```

will retrieve all the elements that are under the container element and that belong to the prefix2NS namespace. In our case this is only the first child of the container element.

Note: The following wildcard operations are not allowed: "*:localName", "local*", "pref:*Name", etc. The asterisk character replaces the local name of an element only.

Navigation when writing: The main purpose of the **Simple XPath** (xpath.expr) parameter is to specify the place where the entry data should be put. . By default this parameter is set to the single wildcard – "*". If the default value is not changed the parser will output a XML with a root element with name DocRoot. You then have the choice to either remove the value of this parameter and have a muti-rooted document or to replace the asterisk with a concrete value.

For example if the following path is set:

```
xpath.expr: /root/container/entry | /otherRoot/otherContainer/moreElements
```

then the parser will create the structure:

```
<root>
 <container>
  <entry>
   /* The Entries go here. */
  </entry>
 </container>
</root>
```

Where the elements root, container and entry are static since they do not belong to any entry passed to the parser as input. Depending on the configuration of the parser these static elements could be written on each cycle (to wrap each entry) or to wrap all the entries.

Note: Only the first path is used and the rest is ignored.

Using asterisks in the **Simple XPath** (xpath.expr) parameter when the XML Parser is in output mode will make the parser consider only the path before the first asterisk. For example the expression:

```
xpath.expr: /root/container/*/entry
```

will be considered as if you specified this expression:

```
xpath.expr: /root/container
```

The only expression that is an exception to the rule is:

```
xpath.expr: *
```

this will be read as if this was specified:

```
xpath.expr: DocRoot
```

You could think of the **xpath.expr** as the parameter that configures the root element(s) only. The parser has the ability to declare a single element that will wrap each entry output as XML. This element could be configured in the **entry.tag** field. If this field is missing a value, then no element, wrapping each entry, will be output. By default this parameter has a value so an additional element will be written to the output stream.The parser also provides a convenient field to configure the name of the element that will contain each value of a simple multi-valued Attribute. This could be configured in the **value.tag** field and by default this is set to *ValueTag* but if it is removed and the parser is asked to output such an Attribute then each value will put in a element with the name "value".

Notes:

1. The **value.tag** parameter is only considered if the **entry.tag** parameter is not empty. If it is empty the values of a multi-valued attribute will not be wrapped.
2. Neither the **entry.tag** nor the **value.tag** support a wildcard and if such is provided then an exception will be thrown as result.

In order to declare some attributes or prefixes in those static elements then you can use the **Static Attributes Declaration** (static.decl) field. If you would like to output the XML used in the "Advanced XML" section you need to use the following configuration:

```
xpath.expr: /root/pref1:container/
static.decl: <root xmlns="defaultNS" xmlns:pref1="prefixNS">
   <pref1:container xmlns:pref2="prefix2NS" attribute1="attrValue1" pref1:attribute2="attrValue2" />
 </root>
```

From this example you can see that the **static.decl** uses XML to markup the attributes and the namespaces that need to be output on the static roots. Note that the XML structure must match the resultant xml structure. This field can also contain information about the Entry Tag element.

If in the above example you add the parameter:

```
entry.tag: Entry
```

you could then add some attributes/namespaces on that level as follows:

```
static.decl: <root xmlns="defaultNS" xmlns:pref1="prefixNS">
 <pref1:container xmlns:pref2="prefix2NS" attribute1="attrValue1" pref1:attribute2="attrValue2" />
  <Entry xmlns:="otherDefaultNS" pref1:attribute3="attrValue3" />
 </pref1:container>
</root>
```

You could define both the entry.tag and value.tag to have prefixes, just like each of the xpath.expr path's elements could have. The difference between the two is that the prefix for the value.tag element must be defined prior to using it. This could be done using the static.decl field or using the hierarchical entry structure provided in this version. Currently it is not possible to include the value.tag element in the static.decl field as the entry.tag is included.

Reading XML

Each time the parser is asked for an entry it reads data from the InputStream and retrieves it as an Entry object. Each element found in the XML is represented from the Attribute class that implements the `org.w3c.dom.Element` interface. Each attribute found in the XML is represented from the Property class that implements the `org.w3c.dom.Attr` interface. Each CDATA found in the XML is represented by the AttributeValue class that implements the `org.w3c.dom.CDATASection` interface.

The StAX implementation of the XLXP project supports neither DTD nor XSD validation, so no validation is possible.

The XML Parser is able to read multi-rooted XML documents as long as it does not have multiple XML declarations. If it has multiple XML declarations, then you can read the XML if and only if the **Ignore repeating XML declarations when reading** check box is checked. This however will affect the performance of the parser since the document will be double-checked for repeating XML declarations.

Notes:

1. Enabling the **Ignore repeating XML declarations when reading** check box will ignore all the XML declarations (except the first one). This means that if a CDATA section contains an XML declaration it will be ignored. To work around this you can fix the CDATA section manually after the entry is retrieved.
2. The XML Parser tries to find the appropriate encoding according to the description in "Character Encoding in the XML Parser" on page 357.

A String representation of the retrieved Entry is available and can be accessed using the `getCurrentEntryAsXMLString()` method.

Writing XML

Each time the parser is asked to write an Entry object in the output stream it will call the `toString()` method on each object that will be included in the XML (the same way the Simple XML Parser behaves). Each entry will be flushed to the output stream separately, and in case of system failure the last flushed entry will have been safely sent.

The parser has the ability to output each Entry in a separate root, by checking the **Multi-rooted Document** option. This will result in a multi-rooted document where each entry has its own static root.

If you are appending the XML to an existing XML then it is useful to check the **No XML declaration when writing** parameter. Checking (enabling) this parameter will instruct the parser to omit the XML declaration that is usually put in the beginning of an XML document.

Character Encoding in the XML Parser

The XML Parser has a parameter **Character Encoding** that you can use to set the name of the encoding. When set the encoding will be used to decode the InputStream passed to the parser during the initialization. When this parameter is other than blank (empty string) then it will be used, regardless of its value. If for example the InputStream is UTF-16BE encoded, has a Byte Order Mark (BOM) at the beginning and the **Character Encoding** parameter is set to "UTF-16BE" then the parser will be able to recognize the BOM sequence and will skip it automatically. If the **Character Encoding** parameter is set to a different encoding (not compatible with the InputStream's encoding) then an exception will be throw which will indicate that an inappropriate encoding is specified.

When you are not sure about the encoding of the InputStream or file then you can let the parser to discover it (if possible). This is the order that the Parser will follow to discover the encoding of the XML if it is not explicitly specified in the configuration (that is, the **Character Encoding** parameter is empty):

1. The Parser will check for a BOM. If it is found then the parser will decode the InputStream using the information provided by that BOM. The recognizable encodings (based on the BOM) are: UTF-8, UTF-16LE, UTF-16BE, UTF-32LE, UTF-32BE.

   Note: The parser does not recognize unusual (reversed) four byte sequences similar to the UTF-32's sequences. In this case an explicit configuration will be required (using the **Character Encoding** parameter).
2. If the InputStream or file does not provide a BOM sequence and no explicit configuration is set then the parser will try to guess the encoding and read the XML declaration's encoding attribute value. The encodings: UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, UTF-32LE and IBM-1047 (EBCDIC variation) will be used to read the specific encoding. If found, that value will be used to decode the rest of the InputStream or file.

   Note: The XML declaration must be set on the first line of the document and must start from the first character.
3. If the **Character Encoding** parameter is not set, no BOM is found and no XML declaration is found (or the XML declaration does not have the encoding attribute) then the parser will use the default encoding which is UTF-8.

We recommend that, if the encoding is known at design time, then it is better to be set it explicitly in the XML Parser's configuration. This will increase the performance of the Parser's initialization process because no lookup for an encoding will be done.

When the parser is initialized for writing (Output Mode) then it expects an explicit assignment of the **Character Encoding** parameter. If no such assignment is done the Parser will use UTF-8 as a default

encoding (UTF-8 with no BOM sequence). If any BOM compatible encoding is explicitly specified (UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, UTF-32LE) then the parser will set a BOM sequence at the beginning of the stream.

Example

The example bundled in *TDI_install_dir*/examples/xmlparser2 demonstrates how IBM Tivoli Directory Integrator is able to work with various XML documents using the capabilities of the XML Parser. Refer to the readme.txt file for more information.

Using XSD Schemas

Predefined XSD schema URI

The configuration of the Tivoli Directory Integrator XML Parser includes a parameter that points to XSD schema(s). When the parser is asked for its schema it reads the schema from the XSD and display it. This however requires that the parser is properly configured. If the navigation path is not set no schema can be retrieved. In that case you will have the ability to read an entry to discover a sample schema.

No XSD provided

If no XSD is provided but the parser is properly configured (that is, the navigation path is set) the Parser will try to extract the Schema Location information from the XML. All schemas found will be checked for the desired element's schema. If no schema is found within the XML document then you will need to read an entry (that is, to read part of the XML) in order to display the content of the returned entry – which is default behavior for all schema querying. However the returned entry's structure might not be the same as another entry that is going to be read on the next cycle. This means that the schema displayed cannot be guaranteed to be complete or valid.

Configuring the Schema

To display the schema of the desired element(s) you must configure the path to it (them) and the path to the corresponding schema(s) (the schema path is optional; see "No XSD provided"). If there are multiple elements and/or schema paths then all paths should be separated by vertical bar – the "|" character. Regardless if schema paths are entered or not the Parser will check for schema locations inside the XML document. The first schema extracted from the XML will be chosen as leading schema; if no schema is returned then the first schema configured by you will be the leading one.

Schemas can be declared with corresponding namespaces when they are configured in the Parser configuration. The configuration is as follows:

```
namespace1 schema1 | namespace2 schema2 | noNamespaceSchema | ...
```

The namespace is used to determine which type in the XSD Schema to which schema file belongs (if it is specified in the schema itself).

Benefits of the leading schema:

This will be the first schema which will be checked for the elements that you entered. If the information is not found in the leading schema then the other schemas entered by you or extracted from the XML are checked. It is advisable to use for leading schema the schema that contains the root element of the XPaths that you have entered.

The library used for schema parsing is slow when it comes to creating the XSD Schema. For this reason all paths are kept in a Map and when a schema is needed for the first time then it is created and kept in the Map in case it is needed later.

The result:

For each element entered in the element's path an Entry will be created. Each entry will contain two attributes – Name and Type. Name will be the name of the element which we are querying. Type will be the corresponding type of the element found in the schema. If the type found in the schema is a primitive type (that is, no definition could be found for it in the provided schemas) then its name is the value of the Type attribute. If the type found is *not* a primitive one (that is, we have found a definition for it in the provided schemas) then as a value of the Type attribute is put a new Entry that will contain attributes with names (the names of all found elements and attributes) and values (the type of the corresponding attribute or element). Again if the type is not primitive a new Entry will be created that will be filled in, in exactly the same manner. This will manifest itself as a tree like structure.

When the schema of all elements is found the entries that are created are put in a Vector object, and this object is returned as the result of the schema querying.

Note: This schema will be displayed flat in the current Configuration Editor. This will be enough for test purposes of the Query Schema functionality.

Indicators:

- Order indicators – There are three possibilities for schema indicators - **all**, **choice** and **sequence**. When one of these indicators is found in the schema file then a property called "#indicator" is set in the result entry. The value of the property is the indicator which is found in the schema. The information inside the Entry must obey the indicator.
- Occurrence indicators – These indicators are set as attributes to the corresponding element. See the Attributes section below.

Attributes:

If a schema element contains any attributes they are kept in a Map in a property "#attributes" which corresponds to the Entry in which the schema of the element will be written.

Example XSD Schema

Schema name: Order.xsd

Schema path: /path/Order.xsd

Contents of Order.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            xmlns="urn:nonstandard:XSD_Schema"
            targetNamespace="urn:nonstandard:XSD_Schema" xmlns:stako="Stako">
    <xsd:element name="order" type="Order" />
    <xsd:complexType name="Order">
        <xsd:all>
            <xsd:element name="user" type="User" minOccurs="1" maxOccurs="1" />
            <xsd:element name="products" type="Products" minOccurs="1" maxOccurs="1" />
        </xsd:all>
    </xsd:complexType>
    <xsd:complexType name="User">
        <xsd:all>
            <xsd:element type="xsd:string" name="deliveryAddress" />
            <xsd:element name="fullname">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                        <xsd:maxLength value="30" />
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:element>
        </xsd:all>
    </xsd:complexType>
    <xsd:complexType name="Products">
```

```
        <xsd:sequence>
            <xsd:element name="product" type="Product" minOccurs="1" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Product">
        <xsd:attribute name="id" type="xsd:long" use="required" />
        <xsd:attribute name="quantity" type="xsd:positiveInteger" use="required" />
    </xsd:complexType>
</xsd:schema>
```

Configuring the Parser to display the User and Products schema:

```
Simple XPath: /Order/User | /Order/Products
XSD Schema Location: /path/Order.xsd
```

Result:

```
[[Name:user, Type:[fullname:[xsd:string:[xsd:maxLength:30]], deliveryAddress:string]],
 [Name:products, Type:[product:[quantity:xsd:positiveInteger, id:xsd:long]]]]
```

This is the `toString()` method representation of the Vector returned as result. Everything is in one row.

XML SAX Parser

The XML SAX Parser is based on the Apache Xerces library. It is used for reading large sized XML documents that the DOM based XML parser won't be able to handle because of memory constraints. It extracts data enclosed within the 'Group tag' supplied in the configuration and creates an Entry with the attributes present in the data. You can specify multiple group tags by separating each tag name with a comma. This will cause the SAX parser to break on any the tags specified. When specifying multiple group tags the SAX parser will use a first-in-win approach where the group tag that was first encountered will be tag that closes the group. As an example, if you have A and B as group tags and the document has a structure where B is a child of A, then A will be the tag closing the entry (as A is found before B and thus takes precedence).

Once a group tag has been found, then any nested occurrence of group tags will have no effect on the current Entry.

If no group tags have been defined, the entire XML document will be returned as a single Entry.

The entry attribute name is composed of surrounding tag names with "@" as the separator. For example, consider the following XML file -

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
 <Entry>
  <Company>
   <Name incorporated="yes">IBM Corporation</Name>
   <Country>USA</Country>
  </Company>
 </Entry>
 <Entry>
  <Company>
   <Name incorporated="no">Smith Brothers</Name>
   <Country>USA</Country>
  </Company>
 </Entry>
</DocRoot>
```

Using "Entry" as the GroupTag, the above XML document would yield two entries as follows -

Entry 1

```
Attribute name: DocRoot@Entry@Company@Name
Attribute value: IBM Corporation
Attribute name: DocRoot@Entry@Company@Name#incorporated
Attribute value: yes
Attribute name:DocRoot@Entry@Company@Country
Attribute value: USA
```

Entry 2

```
Attribute name: DocRoot@Entry@Company@Name#incorporated
Attribute value: Smith Brothers
Attribute name: DocRoot@Entry@Company@Name#incorporated
Attribute value: no
Attribute name:DocRoot@Entry@Company@Country
Attribute value: USA
```

The attribute name may be shortened by specifying a 'Remove Prefix' value in the configuration. For example, a 'Remove Prefix' value of "DocRoot@Entry@Company" in the above example will result in the Entry containing attributes like -

```
Attribute name: Name
Attribute value: IBM Corporation
Attribute name: Name#incorporated
```

```
Attribute value: yes
Attribute name: Country
Attribute value:  USA
...
```

When the Connector is initialized, the XML Parser tries to perform Document Type Definition (DTD) verification if a DTD tag is present. The parser will read multi-valued attributes, although only one of the multi-value attributes will be shown when browsing the data in the Schema tab.

If the XML file has nested entry tags, all Entry tags enclosed within the outermost Entry tag, will be treated as normal XML tags. For example,

```
<entry>
 <entry>
  <company>IBM</company>
 </entry>
</entry>
```

Here the entry will contain the following attribute:

```
attribute name: entry@entry@company
attribute value: IBM
```

Configuration

Group Tag
XML Group tag name(s) that encloses entries. Specify multiple tags by separating each tag name with a comma; or use the root tag if this parameter is not specified (and the entire XML document will be returned as a single Entry).

Remove prefix
Specify the prefix to remove from the attribute names.

Ignore attributes
Asks the parser to ignore attributes of the group tag and its children.

Character Encoding
Character Encoding to be used; the default is UTF-8. Also see “Character encoding.”

Document Validation
Checking this field, requests the validation of the file on basis of the DTD/XSchema used.

Namespace Aware
Checking this field, requests a namespace aware XML parser.

Read Timeout
The time in seconds, after which the parser stops if no data is received.

Detailed Log
If this field is checked, additional log messages are generated.

Character encoding

The default and recommended Character Encoding to use when deploying the XML SAX Parser is UTF-8. This will preserve data integrity of your XML data in most cases. When you are forced to use a different encoding, the Parser will handle the various encodings in the following way:

When reading a file the parser will look for encoding in the following order:

1. If the parser's CharacterSet config parameter is set and is not set to UTF-8, the encoding is set to the value specified in this parameter. However, check #2 is attempted and will overwrite this check if successful when the encoding specified is UTF-32 or UTF-16.
2. The XML you are parsing is checked for the existence of an encoding attribute from the XML declaration. If the encoding attribute from the XML declaration is found, this value will be used.

3. The default encoding of the JRE is used if none of the above are true (Normally, UTF-8)

See also

"XML Parser" on page 351,

"Simple XML Parser" on page 347,

"XSL based XML Parser" on page 365.

XSL based XML Parser

Introduction

The XSL based XML DOM Parser enables Tivoli Directory Integrator to parse XML documents in any format using the XSL supplied by the user, into attribute value pairs, stored in the entry object. The XSL based parser is required to facilitate reading of any kind of XML format. Particularly, when the user needs only a specific chunk of the XML he can write an XSL for picking the required chunk. The parser will create an in-memory parse tree to represent the input XML and the Tivoli Directory Integrator internal format. The XSL transforms the DOM Document generated from input XML, and produces an output DOM for the Tivoli Directory Integrator internal format. It uses the javax transformation libraries to carry out transformations.

Configuration

The XSL based DOM XML Parser provides the following parameters:

Use input XSL file
Check box to indicate whether to use input XSL file or use the XSL keyed in (in the *Input XSL* field)

Input XSL File Name
The input XSL file that contains template matching rules for transforming user XML to Tivoli Directory Integrator internal format

Input XSL
Editable area to allow the user to key in or paste the entire input XSL.

Use output XSL file
Check box to indicate whether to use output XSL file or use the XSL keyed in (in the *Output XSL* field).

Output XSL File Name
The output XSL file that has template matching rules for transforming Tivoli Directory Integrator internal format back to user XML

Output XSL
Editable area to allow the user to key in or paste the entire output XSL.

Character Encoding
The character encoding to use when reading or writing; the default is UTF-8.

This Parser extends the Simple XML Parser; therefore, the same notices with regards to Character Encoding apply.

Omit XML Declaration
If checked, omit XML declaration header in output stream.

Document validation
if checked, request a DTD/XSchema validating XML parser.

Namespace aware
If checked, request a namespace aware XML parser.

Indent Output
If checked, causes the output to be neatly indented, improving human readability. If your output is going to be processed by another program, this option is best left off.

Detailed log
Specifies whether detailed debug information is written to the log.

Using the Parser

The parser can be used with the Filesystem Connector in *Iterator* or *AddOnly* mode. The XSL based DOM XML parser requires the user to specify:

- The input XSL file (when used in a Filesystem Connector in Iterator mode): to transform XML to Tivoli Directory Integrator internal format.
- The output XSL file (when used in a Filesystem connector in AddOnly mode): to transform Tivoli Directory Integrator internal format back to the original format.

In an XSL transformation, an XSLT processor reads both an XML document and an XSLT style sheet. Based on the instructions the processor finds in the XSLT style sheet, it outputs a new XML document or fragment thereof. The parser will do the basic validation of the XSL files for authenticity. The parser also has optional Document and namespace validation of the file supplied by the Connector. The parser can be used in conjunction with the filesystem connector. The parser will support reading as well as writing, in the sense that XML files can be read and written to in a format specified by the respective XSL. The following optional validations are provided:

- Document validation
- Namespace aware

Tivoli Directory Integrator Internal Format

```
<DocRoot>
 <Entry>
  <attribute_name>
   <value_tag>attribute_value</value_tag>
   <value_tag>attribute_value</value_tag>
   <value_tag>attribute_value</value_tag>
  </ attribute_name>
  <attribute_name>
   <value_tag>attribute_value</value_tag>
  </ attribute_name>
     -
     -
     -
 </Entry>
 <Entry>
     -
     -
     -
 </Entry>
 -
</DocRoot>
```

Example

Input XML: birds.XML

```
        <?XML version="1.0" encoding="UTF-8"?>
        <Class>
        <Order Name="TINAMIFORMES">
                <Family Name="TINAMIDAE">
                    <Species Scientific_Name="Tinamus major">  Great Tinamou.</Species>
                    <Species Scientific_Name="Nothocercus">Highland Tinamou.</Species>
                    <Species Scientific_Name="Crypturellus soui">Little Tinamou.</Species>
                    <Species Scientific_Name="Crypturellus cinnamomeus">Thicket Tinamou.</Species>
                    <Species Scientific_Name="Crypturellus boucardi">Slaty-breasted Tinamou.</Species>
                    <Species Scientific_Name="Crypturellus kerriae">Choco Tinamou.</Species>
                </Family>
            </Order>
        <Order Name="GAVIIFORMES">
                <Family Name="GAVIIDAE">
                    <Species Scientific_Name="Gavia stellata">Red-throated Loon.</Species>
                    <Species Scientific_Name="Gavia arctica">Arctic Loon.</Species>
                    <Species Scientific_Name="Gavia pacifica">Pacific Loon.</Species>
                    <Species Scientific_Name="Gavia immer">Common Loon.</Species>
```

```
            <Species Scientific_Name="Gavia adamsii">Yellow-billed Loon.</Species>
          </Family>
       </Order>
    </Class>
```

Input XSL: birds.XSL

```
<?XML version="1.0" ?>
<XSL:stylesheet xmlns:XSL="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <XSL:output method="XML" indent="yes" />
<XSL:template match="Class">
 <DocRoot>
  <XSL:for-each select="Order">
     <XSL:variable name="order"><XSL:value-of select="@Name" />
     </XSL:variable>
     <XSL:for-each select="Family">
     <Entry>
      <Attribute name="Order">
   <Value><XSL:value-of select="$order" /></Value>
  </Attribute>
     < Attribute name="Family">
   <Value><XSL:value-of select="@Name" /></Value>
     </Attribute>
     <Attribute name="Species">
      <XSL:for-each select="Species">
             <Value><XSL:value-of select="." /></Value>
      </XSL:for-each>
     </Attribute>
     </Entry>
  </XSL:for-each>
    </XSL:for-each>
 </DocRoot>
</XSL:template>
</XSL:stylesheet>
```

birds.xsl transforms birds.xml to Tivoli Directory Integrator internal format from entry object with attribute value pairs, can be formed.

See also

"Simple XML Parser" on page 347

The XML Bible (the chapter on XSL)
http://www.ibiblio.org/xml/books/bible2/chapters/ch17.html

W3C Document Object Model
http://www.w3.org/DOM/

Effective XML processing with DOM and XPath in Java
http://www.ibm.com/developerworks/xml/library/x-domjava/

User-defined parsers

In addition to the parsers already provided with the installation of IBM Tivoli Directory Integrator, you can write your own parsers and add them to the system.

One example of a user-defined parser, capable of parsing Regular Expressions, is provided in the Examples directory. Go to the *root_directory*/examples/regexp_parser directory of your IBM Tivoli Directory Integrator installation. This particular example was written in Java.

Another example of a user-defined parser can be found in *root_directory*/examples/script_parser, which shows how to write a parser using scripting. See "JavaScript Parser" in the *IBM Tivoli Directory Integrator V7.1 Users Guide* for a further explanation of this example.

Chapter 4. Function Components

Function Components (FC) are, besides of Connectors and Parsers, another type of building block that make up the IBM Tivoli Directory Integrator. Function Components are similar in scope to a Connector, with the difference that the latter are datasource specific whereas a Function Component is not. Rather, it is an AssemblyLine Component that facilitates wrapping of custom logic and external methods, and presents a user friendly "connector-like" user interface in the Configuration Editor (CE).

Also, a Function Component is modeless; that is, in order to configure a Function Component in your AssemblyLine you don't have to specify in which mode it is supposed to operate. It will do its work in the `perform()` method whenever it is called by the AssemblyLine.

Many of the components described below provide the means to build a complete (both client-side and server-side) web service solution in the modular Tivoli Directory Integrator web service architecture.

The Function Components provided with the IBM Tivoli Directory Integrator 7.1 are:

- "AssemblyLine Function Component" on page 385
- "Axis EasyInvoke Soap WS Function Component" on page 421
- "Axis2 WS Client Function Component" on page 417
- "Axis Java To Soap Function Component" on page 405
- "Axis Soap To Java Function Component" on page 415
- "Castor Java to XML Function Component" on page 373
- "Castor XML to Java Function Component" on page 375
- "CBE Function Component" on page 395
- "Complex Types Generator Function Component" on page 425
- "Delta Function Component" on page 427
- "Function Component For SAP ABAP Application Server" on page 449
- "InvokeSoap WS Function Component" on page 411
- "Java Class Function Component" on page 389
- "Memory Queue Function Component" on page 402
- "Parser Function Component" on page 391
- "Remote Command Line Function Component" on page 431
- "Scripted Function Component" on page 393
- "SendEMail Function Component" on page 399
- "WrapSoap Function Component" on page 409
- "XMLToSDO Function Component" on page 377
- "SDOToXML Function Component" on page 381
- "z/OS TSO/E Command Line Function Component" on page 437

Castor Java to XML Function Component

Processing complex and custom data types is often a requirement for various XML solutions, for example Web Services.

The existence of a self-contained Java-to-XML and XML-to-Java binding functionality in the IBM Tivoli Directory Integrator provides the ability to process complex/custom data types independently of a Web Service toolkit. In particular this means that there is an option to deal with possible binding limitations of various Web Service toolkits.

Castor Overview

Castor is an open source data binding framework providing access to the data defined in an XML document through an object data model.

Castor can marshal almost any "bean-like" Java object to and from XML. The process of marshalling/unmarshalling can use Castor's default introspection model (an implementation based on Java reflection where Castor decides how to marshal and unmarshal data), but this process can also be controlled and customized by the use of Castor XML Mapping Files that define mapping rules.

From an IBM Tivoli Directory Integrator perspective, you can create XML Mapping Files and specify how custom data is mapped to and from XML.

With Castor you can process an XML document not specially designed for Castor by skipping parts of the XML you are not interested in. A limitation here is that Castor cannot skip an XML node and process a node belonging to the subtree of the skipped node. This limitation is a serious inconvenience when you want to extract random parts of a big and complex XML document (cases which might be expected in the real world) – that is why the IBM Tivoli Directory Integrator Castor Function Components provide the ability to specify certain parts of the XML through XPath queries.

The CastorJavaToXML Function Component uses Castor 0.9.5.4. Documentation and information for the Castor library can be obtained from the Castor project web site: http://www.castor.org/

Configuration

Parameters

Castor Mapping File
: An XML Mapping File (as defined by the Castor syntax) that defines how Java or *Entry* objects are serialized into XML.

  The mapping file as specified by this parameter should always include the mapping rules defined in the *TDI_install_dir*/etc/di_castor_mapping.xml file. This means that either you must specify "etc/di_castor_mapping.xml" as value of this parameter or make sure that the mapping file specified contains these rules (for example by using Castor's "include" clause to include "di_castor_mapping.xml" rules in another mapping file).

XML Root Element
: The name of the root element of the generated XML; if left empty the root element is named "Entry".

Use Attribute Names
: If this is checked, the names of the Attributes are used as XML element names, otherwise the XML elements are named as specified in the Mapping File. This parameter is only taken into account in Entry mode.

Return XML as
: This drop-down list specifies the return type - only taken into account when the input object is not an object of type *Entry*. Valid values are **String** and **DOMelement**.

Detailed Log
Check to generate additional log messages.

Comment
Your own comments go here.

Using the FC

The CastorJavaToXML Function Component creates an XML document from a Java object or an *Entry* object.

This Function Component can operate both with *Entry* objects and with custom Java objects.

When the Function Component is passed an *Entry* object on input, it will return an *Entry* object. This mode of operation is called **Entry** mode.

When passed a Java object which is not an Entry object, the Function Component will serialize the object passed using Castor serialization and this will be the result XML. This mode of operation is called **non-Entry** mode.

Entry mode

- In **Entry** mode each Attribute of the Entry passed on input is marshalled and placed under the root of the resulting XML element.
- If the **Return XML as** parameter is set to **DOMelement** the resulting Entry contains one attribute named "*xmlDOMElement*" and its value is the marshaled XML element as a "`org.w3c.dom.Element`" object.
- If the **Return XML as** parameter is set to **String** the resulting Entry contains one attribute named "*xmlString*" and its value is the serialized XML element as a "`java.lang.String`" object.

Non-Entry mode

- If the **Return XML as** parameter is set to **DOMelement** the resulting XML element is returned as a "`org.w3c.dom.Element`" object.
- If the **Return XML as** parameter is set to **String** the XML element is returned as a "`java.lang.String`" object.

Castor XML to Java Function Component

The CastorXMLtoJava Function Component is the mirror-image counterpart to the "Castor Java to XML Function Component" on page 373, and the same section on "Castor Overview" on page 373 applies.

Specifically, the CastorXMLtoJava FC creates an *Entry* or a general Java object from an XML document, and it provides the option to get data from certain parts of the XML tree when deserializing the XML document.

In addition to the Castor mapping mechanism which specifies how to build a Java object (possibly of a custom Java class) from an XML node/subtree, this Function Component provides its own logic to specify how to populate Entry Attributes from an XML document. By using XPath queries you can specify which parts of the XML document will be passed to the Castor APIs for deserializing.

This approach both provides ease of use in the IBM Tivoli Directory Integrator context and gives more power when processing custom XML documents, for example XML documents generated by other systems. Through the XPath queries you are able to specify which parts of the XML you are interested in (and get them directly into Entry Attributes) and which are irrelevant for your process and should not be processed. In addition, the writing of the Castor XML Mapping Files is facilitated since you will only have to write mapping rules for the parts of the XML document you are interested in and not for the whole XML document.

The CastorXMLToJava Function Component uses Castor 0.9.5.4. Documentation and information for the Castor library can be obtained from the Castor project web site: http://www.castor.org/

Configuration

Parameters

Castor Mapping File

An XML Mapping File (as defined by the Castor syntax) that defines how XML is mapped to Java objects.

The mapping file as specified by this parameter should always include the mapping rules defined in the *TDI_install_dir*/etc/di_castor_mapping.xml file. This means that either you must specify "etc/di_castor_mapping.xml" as value of this parameter or make sure that the mapping file specified contains these rules (for example by using Castor's "include" clause to include "di_castor_mapping.xml" rules in another mapping file).

Attribute Specification

Each line specifies a single Attribute in the format: <AttributeName>,<XPath query>[,<type>] . This parameter is only taken into account when the Function Component is used in Entry mode. Within each line,

<AttributeName>

specifies the name of the Entry Attribute;

<XPath query>

specifies which part(s) of the XML to unmarshal and assign as value to this Attribute.

<type>

must be used whenever the type of the Attribute is not a complex Java class, but one of the following basic data types: *string*, *date*, *boolean*, *integer*, *long*, *double*, *float*, *big-decimal*, *byte*, *short*, *character*, *strings* (array of strings), *chars* (array of chars), *bytes* (array of bytes). In these cases the user must specify the type, because of a limitation in Castor – Castor cannot handle these types when they map to standalone objects (instead of being members of other objects) and so the Function Component needs to know the type and take special actions to make Castor produce the correct object.

XML Input as
This drop-down list specifies whether the Function Component will accept the input XML data in the form of a **DOMElement** object or as a **String**.

Detailed Log
Check to generate additional log messages.

Comment
Your own comments go here.

Using the FC

The CastorXMLToJava Function Component creates an *Entry* or a general Java object from an XML document, and can operate both with Entry objects and with custom Java objects.

When the Function Component is passed an Entry object on input, it will return an Entry object. This mode of operation is called **Entry** mode.

When the Function Component is passed an object that is not an Entry on input (**String** or a **DOMelement**) it returns the raw Java object as it is unmarshalled by Castor. This mode of operation is called non-Entry mode.

Entry mode

- If the **XML Input as** parameter specifies **DOMelement**, the Function Component will expect on input an Entry with an Attribute named "*xmlDOMElement*" with value of type "`org.w3c.dom.Element`".
- If the **XML Input as** parameter specifies **String**, an Entry with an Attribute named "*xmlString*" and value of type "`java.lang.String`" is expected on input.
- The output generated is an *Entry* whose Attributes are the unmarshalled XML elements as specified by the **Attribute Specification** parameter and the mapping file.

Non-Entry mode

- If the **XML Input as** parameter specifies **DOMelement**, the Function Component will expect on input a "`org.w3c.dom.Element`" object.
- If the **XML Input as** parameter specifies **String**, a "`java.lang.String`" object is expected on input.

XMLToSDO Function Component

The EMF XMLToSDO Function Component converts an XML document to SDO objects connected in a tree-like structure resembling the XML structure.

Note: The XMLToSDO Function Component is deprecated in the IBM Tivoli Directory Integrator 7.1 release and will be removed in a future version.

For each XML element an XML Attribute Data Object is created. Tivoli Directory Integrator Entry Attributes are then created for some of the Data Objects. The name of the Entry Attribute consists of the names of the ancestor elements of the element the Tivoli Directory Integrator Attribute represents. Subsequent XML element names are separated by the "@" character. When an XML attribute is represented, the name of the XML attribute is appended to the name of the XML element using the "#" symbol as a separator.

All Attribute names start with the "DocRoot" text which represents the XML root. There are two types of Entry Attribute values:

- The standard Java wrapper when the XML element/attribute value is a primitive type (java.lang.String, java.lang.Integer, java.lang.Boolean, etc.)
- A Service Data Object when the XML element is a complex XML structure. This will be an object of type org.eclipse.emf.ecore.sdo.EDataObject.

XML elements that have a common parent element are called siblings. Sibling elements with the same name are grouped in a multi-valued Tivoli Directory Integrator Attribute Entry.

Note: Attributes are not created for XML elements with an ancestor element that has a sibling with the same name. Those can only be accessed through the multi-valued Attribute representing the siblings.

Example

This example illustrates how the following XML file is processed by the EMF XMLToSDO Function Component:

```
<?xml version="1.0">
<database name="Persons">
 <description>This is a sample database</description>
 <person>
  <name>Ivan</name>
  <age>21</age>
 </person>
 <person>
  <name>George</name>
  <age>32</age>
 </person>
</database>
```

When the EMF XMLToSDO Function Component processes the example XML File, an entry with the following Attributes is created:

- DocRoot – a Service Data Object representing the XML root
- DocRoot@database - a Service Data Object representing the "database" XML element
- DocRoot@database#name – a java.lang.String object representing the "name" XML attribute of the "database" XML element.
- DocRoot@database@description - a java.lang.String representing the "description" XML element (which is a child of the "database" XML element).
- DocRoot@database@person – multi-value attribute whose values are Service Data Objects representing the individual "person" XML elements.

"DocRoot@database@person@name" is not a valid Tivoli Directory Integrator Attribute in this case because more than one person XML element exists in the XML document at the same level.

The EMF XMLToSDO Function Component provides an option to use namespace prefixing. Namespace prefixing option specifies that all XML element names part of the Entry Attribute name will be prefixed with the corresponding namespace; for example: "DocRoot@namespace1:database@namespace2:person".

Configuration

XSD File

Specifies the location of the XML Schema (XSD) File. The XML Schema File is used in the process of reading the XML document, in the generation of an EMF Ecore Model and in the Discover Schema functionality. This parameter is required.

The extension of the XML Schema File specified must be ".xsd".

Use Namespaces

Specifies whether XML elements and attributes namespaces will be set in the generated Entry Attribute names. When this parameter is checked XML elements and attributes will be prefixed either with a prefix defined in the "namespaceMap" parameter or with the namespace URI if no prefix is defined in "namespaceMap".

This parameter also specifies whether the Discover Schema functionality will use XML namespaces to prefix the Entry Attribute names.

Namespace Map

Defines a mapping between namespace prefixes and namespace URIs. Each pair is specified on a new line. The prefix is delimited from the URI with an equal sign, for example "ibm=http://www.ibm.com". Preceding and trailing white space for both the prefix and the URI is ignored.

This parameter is only taken into account if the "useNamespaces" parameter is set to true.

Input XML as

Specifies the type of the input XML document. It can be a java.lang.String object or org.w3c.dom.Element object.

Encoding

Specifies the character set to use for encoding converting to and from XML. If left blank the default system charset is used.

Debug

Turns on debug messages.

Migration

The EMF XMLToSDO and SDOToXML Function Components are not compatible with the Tivoli Directory Integrator 6.0 Castor Function Components. Any solution which uses the Castor Function Components needs to be re-implemented in order to work with the EMF XMLToSDO and EMF SDOToXML Function Components. The Castor XML To Java Function Component supports a mapping file. This mapping file can be used to specify how a complex custom XML is to be parsed and converted to a complex custom Java object. This feature is not supported by the EMF XMLToSDO Function Component. By following the next broad guidelines, a Tivoli Directory Integrator 6.0 configuration can be re-implemented to work with the EMF XMLToSDO Function Component:

1. Insert the EMF XMLToSDO Function Component into an AssemblyLine.
2. Set its parameters accordingly.
3. Insert a Script Component into the AssemblyLine right after the EMF XMLToSDO Function Component.

4. Write Javascript code in this Script Component, which extracts the desired data from the SDO DataObject returned by the EMF XMLToSDO Function Component and populates the custom Java Object needed.

The Castor XML To Java Function Component used to support a mechanism which allowed a specific portion of the XML to be mapped to Entry Attributes. The EMF XMLToSDO Function Component does not support this feature. The EMF XMLToSDO Function Component always parses and maps the entire XML to Entry Attribute. By using the Input Attribute Map of the EMF XMLToSDO Function Component, however, only the desired Attributes can be mapped thus emulating the behavior of the Castor XML To Java Function Component.

The Castor Java To XML Function Component used to support a mapping file, which could be used to specify how to serialize a complex Java object into XML (element/attribute names, etc.). The EMF SDOToXML Function Component serializes into XML based on an XML Schema file, that is, the names of elements/attributes, etc. are specified in the XML Schema file specified as a Function Component parameter.

SDOToXML Function Component

The EMF SDOToXML Function Component converts Service Data Objects to XML. This component uses an XML Schema definition to build an Ecore model.

Note: The SDOToXML Function Component is deprecated in the IBM Tivoli Directory Integrator 7.1 release and will be removed in a future version.

The Function Component receives an Entry whose Attributes represent an XML document. The types of the Entry Attribute values are either Java classes representing primitive types or Service Data Objects (org.eclipse.emf.ecore.sdo.EDataObject) representing complex XML elements.

The Entry Attribute names describe the XML hierarchy in exactly the same manner as the EMF XMLToSDO Function Component constructs Attribute names. All Attribute names start with "DocRoot" which represents the XML root. Subsequent elements down the XML hierarchy are separated with the "@" character. If the Tivoli Directory Integrator Entry Attribute represents an XML attribute the "#' character is used to separate the name of the XML attribute from the name of the XML element containing this attribute.

It is possible that the Tivoli Directory Integrator Entry passed contains only Entry Attributes corresponding to the real data. For example, the Entry may contain an Attribute "DocRoot@database@person" without containing an Attribute "DocRoot@database" – the EMF SDOToXML Function Component will automatically create the "database" XML element in the XML document it builds. The EMF SDOToXML Function Component uses the XML Schema to track and create all XML elements that are ancestors of the specified XML element or attribute.

It might happen that the Entry contains Attributes specifying XML elements that are contained in other XML elements specified by Entry Attributes, for example the Entry contains both "DocRoot@database@person" and "DocRoot@database" Attributes. In this case the Attributes are processed starting from the one that is closest to the root, continuing with the one closest to it and so on – the last one will be the most specific XML element that is contained in all the other. This order of processing provides the option to change specific details in a bigger XML context.

For example, if you want to change just the "DocRoot@database@person" element but you want to leave the other parts of the "DocRoot@database" element untouched, you might read the XML document with the EMF XMLToSDO Function Component, map the "DocRoot@database" attribute and provide it to the EMF SDOToXML Function Component as is. Then you will also provide the "DocRoot@database@person" Attribute that contains the specific updates you want to make on the "person" XML element(s). The EMF SDOToXML Function Component will first process the "DocRoot@database" applying all the content to the resulting XML and it will then override the "person" child of the "database" element with whatever is provided in the "DocRoot@database@person" Entry Attribute.

In case a multi-valued Attribute is provided together with an Attribute specifying a child or other successor of that element, the function Component will signal an error (throw exception) because it cannot be determined to which of the sibling XML elements, this successor applies. For example, if "DocRoot@database@person" is provided and contains two values (thus specifying two XML "person" elements at the same level) and also "DocRoot@database@person@name" is provided, the Function Component would not know to which "person" element of the two existing this "name" element applies to. The names of the elements in the Entry Attribute can be XML namespace prefixed.

The names of the elements are prefixed with the namespace URI or with the prefixes defined in the "namespaceMap" parameter.

For example, in order to construct the following XML document:

```
<?xml version="1.0"?>
<database xmlns="www.ibm.com" xmlns:tmp="www.tmp.com" name="employees">
 <person>
  <name>Ivan</name>
  <tmp:age>21</tmp:age>
 </person>
</database>
```

the following Tivoli Directory Integrator Entry can be passed to the EMF SDOToXML Function Component:

- DocRoot@ibm:database#ibm:name
- DocRoot@ibm:database@ibm:person@ibm:name
- DocRoot@ibm:database@ibm:person@www.tmp.com:age

The namespace prefixes used assume that the "namespaceMap" parameter contains the "ibm" prefix set to "www.ibm.com" and no namespace prefix is defined for "www.tmp.com" (that is why it is used directly in the Attribute name).

Configuration

XSD File
: The parameter specifies the location of the XML Schema File. The XML Schema File is used in the process of generating the XML document and in the Discover Schema functionality. This parameter is required. The extension of the XML Schema File specified must be ".xsd".

Use Namespaces
: Specifies whether the Discover Schema functionality will use XML namespaces to prefix the Entry Attribute names. When this parameter is checked XML elements and attributes will be prefixed either with a prefix defined in the "namespaceMap" parameter or with the namespace URI if no prefix is defined in "namespaceMap".

Namespace Map
: Defines a mapping between namespace prefixes and namespace URIs. Each pair is specified on a new line. The prefix is delimited from the URI with an equal sign, for example "ibm=http://www.ibm.com". Preceding and trailing white space for both the prefix and the URI is ignored.

Return XML as
: Specifies the type of the XML document that will be returned by the Function Component. It can be a java.lang.String object or an org.w3c.dom.Element object.

Encoding
: Specifies the character set to use for encoding converting to and from XML. If left blank the default system charset is used.

Debug
: Turns on debug messages.

Using the FC

Migration

The EMF XMLToSDO and SDOToXML Function Components are not compatible with the Tivoli Directory Integrator 6.0 Castor Function Components. That is why any solution which uses the Castor Function Components needs to be re-implemented in order to work with the EMF XMLToSDO and EMF SDOToXML Function Components. The Castor XML To Java Function Component used to support a mapping file. This mapping file could be used to specify how a complex custom XML is to be parsed and converted to a complex custom Java object. This feature is not supported by the EMF XMLToSDO Function Component. However by following the next broad guidelines such a Tivoli Directory Integrator 6.0 configuration can be re-implemented to work with the EMF XMLToSDO Function Component:

1. Insert the EMF XMLToSDO Function Component into an AssemblyLine.
2. Set its parameters accordingly.
3. Insert a Script Component into the AssemblyLine right after the EMF XMLToSDO Function Component.
4. Write Javascript code in this Script Component, which extracts the desired data from the SDO DataObject returned by the EMF XMLToSDO Function Component and populates the custom Java Object needed.

The Castor XML To Java Function Component used to support a mechanism which allowed a specific portion of the XML to be mapped to Entry Attributes. The EMF XMLToSDO Function Component does not support this feature. The EMF XMLToSDO Function Component always parses and maps the entire XML to Entry Attribute. By using the Input Attribute Map of the EMF XMLToSDO Function Component, however, only the desired Attributes can be mapped thus emulating the behavior of the Castor XML To Java Function Component.

The Castor Java To XML Function Component used to support a mapping file, which could be used to specify how to serialize a complex Java object into XML (element/attribute names, etc.). The EMF SDOToXML Function Component serializes into XML based on an XML Schema file, that is, the names of elements/attributes, etc. are specified in the XML Schema file specified as a Function Component parameter.

AssemblyLine Function Component

The AssemblyLine Function Component (AL FC) wraps the calling of another AssemblyLine into a Component, with some controls on how the other AssemblyLine is executed and what to do with a possible result.

The AL FC uses the Server API to call and manage the ALs. The component establishes a server connection to the Server API through RMI and creates a session with the server.

Configuration

AssemblyLine
A drop-down list of pre-defined AssemblyLines that could be the target of this FC.

Server The Tivoli Directory Integrator Server on which the AssemblyLine should be run. Use "*Local*" or blank for internal server or *hostname[:port]* for remote server.

Config Instance
Specify the config instance when using a remote server.

Execution Mode
A drop-down list of three possible modes:

Run and wait for result
Result can be picked up as described in the JavaDocs for this FC; this typically involves calling the FC with an empty Entry object. The returned Entry object contains the reference to the target AL in its "value" attribute.

Run in background
This starts the AssemblyLine asynchronously, and does not wait for any results.

Manual (cycle mode)
Run the AssemblyLine for 1 cycle only.

Custom Keystores
Check to use the "api.remote.server." java properties instead of standard "javax.net.ssl." properties for keystore configuration.

Use TCB Attributes
When checked the FC will interpret attributes with a "$tcb." prefix as parameters to the TCB and remove them from the entry.

Simulate
Check this to run the called AssemblyLine in Simulate mode, which implies that the called AssemblyLine will make use of its Simulation Config when interacting with external systems.

Share Logging
If checked, the called AssemblyLine will use the same logging as this Component.

Operation
Choose from available exposed Operations defined in the target AssemblyLine. The **Query** button will attempt to retrieve the schema (Input or Output Attributes) from the target AL; see “AssemblyLine Connector” on page 14 and Appendix D, “Creating new components using Adapters,” on page 575 for more information.

AssemblyLine Parameters
When the appropriate AL is chosen this field will provide access to that AL's initialization parameters.

Note: This field will stay empty if the remote AL does not have defined initialization parameters in its configuration, that is, the *$initialization* schema.

Detailed Log
When checked, generates additional log messages.

Comment
Your own comments go here.

Using the FC

This FC provides a handler object for calling and managing AssemblyLines on either the local or a remote Server.

You configure this FC by choosing the AL to call, the Server on which this AL is defined and should run on (blank or "local" indicating that the AL runs on this Server which is running the FC), as well as the Config Instance that the AL belongs to. Again, a blank parameter value means that this AL is in the same Config Instance as the one containing the FC itself.

You also choose the Execution Mode (see "AL Cycle Mode" in *IBM Tivoli Directory Integrator V7.1 Users Guide* for more information). Although there are three Execution Modes (Run and wait for completion, Run in background and Manual cycle mode), the first two options are the standard methods of starting an AL from script with or without calling the AL `join()` method.

These first two modes cause the target AL to run on its own (stand alone) in its own thread. The third mode, cycle mode, means that the target AL is controlled by the FC which will execute it one cycle at a time for each time the FC is invoked. When the FC runs an AssemblyLine in stand-alone mode, the FC keeps a reference to the target AL – just like you get when you call `main.startAL()`. The FC can also return the status of the running/terminated ALs. You obtain this status by calling the FC's perform() method with a null or empty Entry parameter. The returned Entry object contains the reference to the target AL in an attribute called "value". If you pass a null value to the FC, the return value is the actual reference to the target AL (again, like making a `main.startAL()` call).

You can also call the FC with specific string command values to obtain info about the target AL:

perform("target")	returns the object reference of the target AL.
perform("active")	returns either "active", "aborted" or "terminated" depending on the target AL status.
perform("error")	returns the java.lang.Exception object when the status is "aborted".
perform("result")	returns the current result Entry object.
perform("stop")	tries to terminate an active target AL, and will throw an error if the call does not succeed.

Note that if you have specified the "Run and wait for completion" Execution Mode, then each call to perform() starts the target AL and returns the complete status for the execution (for example, reference to the target as well as status and error object). In this case, the initialize() method does NOT start the target AL as it does in all other cases. When the FC is called in this mode with an Entry object, the Entry object can contain one or more of the above keywords in an attribute called command (as described in the list above, and concatenated in a comma-separated list). The returned Entry object is then populated with the same values as described above. So, rather than calling perform() several times with each desired command, you can create an Entry with all keywords as attributes in the Entry object and get away with one call to perform():

```
var e = system.newEntry();
e.setAttribute("command", "target, status");
// In this example, fc references a Function Interface.
// If this was an AL Function instead, then fc.callreply(e)
// would be done.
var res = fc.perform(e);
task.logmsg("The status is: " + res.getString("status"));
```

When the FC runs an AL in manual mode, each call with an Entry object causes one cycle to be executed in the target AL. The returned Entry object is the work entry result at the end of the cycle. When the target AL has completed, a null entry is returned. If the cycle execution causes an error, then that error is re-thrown by the FC (so you should use a try-catch block in your script).

The target AL can be supplied with parameters, in two different ways.

By means of a Task Call Block (TCB)
: You can use the method fc.getTCB() and set parameters in the returned TCB object. This object will be used the next time an AssemblyLine is started by this FC. Only connector parameters should be set in the returned TCB as this FC will potentially overwrite the runmode and initial work entry.

By means of special attributes
: Another way to set TCB parameters is by using the output attribute map where variables should be defined with the specific prefix "$tcb.". When these attributes are found in the entry they will be moved to the TCB and removed from the entry. This will only work when the FC runs an AssemblyLine each time the FC is called (that is, run and await completion).

The Query ("Quick Discovery") button in the FC Input and Output Map tabs will try the following methods for determining the schema of the AL to be called:

1. If the **Operation** parameter is set the FC will get any attributes that are defined in the Input and Output maps of that operation.
2. If the AL has a defined schema (AL Call/Return tab), then this will be used.
3. Otherwise the FC examines the Input and Output maps of all Connectors in the AL to be called in order to "guess" its schema.

In the target AL you can define an operation called "querySchema"; if this is the case the Input and Output attributes of this operation are used to supply the AL FC (or the AssemblyLine Connector) with the schema.

See also

“AssemblyLine Connector” on page 14,
Appendix D, “Creating new components using Adapters,” on page 575.

Java Class Function Component

IBM Tivoli Directory Integrator 7.1 (Tivoli Directory Integrator) allows you to use Java objects in your script code to perform specific operations not provided directly by Tivoli Directory Integrator. Because calling methods of Java objects when the Java object must be constructed and parameters mapped to proper classes can be difficult, the Java class Function Component makes using Java objects in your scripts easier. The Java Class Function Component allows you to choose a Java class and method through the Config Editor and performs the conversion and mapping of parameters to the method.

Schema

The schema for the Java Class Function Component is dynamic and reflects the chosen Java class and method. The Function Component also performs dynamic conversion of parameters to match the signature of the target Java class/method.

Parameter Conversion

Parameter conversion is performed for the most common types. However, it is beyond the scope of this FC to provide conversion for all potential Java class objects. For unsupported objects you must explicitly create these before invoking the Java Class Function Component. Below is a table of objects that the Java Class Function Component will recognize for parameter conversions.

Table 66.

Parameter type	Notes
Integer	Both object and primitive type
Long	Both object and primitive type
Double	Both object and primitive type
Float	Both object and primitive type
Short	Both object and primitive type
Byte	Both object and primitive type
Character	Both object and primitive type
Boolean	Both object and primitive type
Date	Only conversion from default date format as defined by DateFormat
String	

In addition to these types, the Java Class Function Component will also attempt conversion into primitive arrays and java.util.Collection objects.

Configuration

The Java Class Function Component uses the following parameters:

JAR/Class File
This parameter specified the file in which the Java class is found.

Java class
Specifies the fully qualified name of the Java class. This parameter is required.

Method
Specifies the method to call in the Java class.

Parser Function Component

The Parser FC wraps a Parser into an AssemblyLine Component, such that it can be inserted anywhere in the AssemblyLine data flow.

Multiple instances of Parser FCs could aid in decoding two or more layers of protocol.

Configuration

Operation Mode
Operation mode of the Parser: Read an Entry from parser, Write an Entry to parser (returning result)

Returns result as String
Check to have the function return a String object instead of a *Bytearray* object

Character Set
The character set to use if value is returned as String object. The default is UTF-8.

Detailed Log
When checked, generates additional log messages.

Comment
Your own comments go here.

A Parser Function Component also has a **Parser** tab. Using the Parser tab, you can select and configure the Parser you want to use to interpret or generate data stream records.

Using the FC

This FC allows you to select a Parser and then set its mode to either input (**Read**) or output (**Write**).

In **Read** mode, you must provide an attribute (in the Output Map) called "*value*" which is either a string, a File, a Reader or a java.io.InputStream object to be used as input for the Parser. The FC will return an *Entry* object (conn) with the parsed attributes, which are then available for your Input Map.

In **Write** mode the FC takes an Entry with the attributes passed in by the Output Map and applies the Parser to that Entry, providing the return bytestream in the Attribute named "*value*". This Attribute is a java.lang.String if you select the **Return result as String** checkbox in the Config tab; otherwise it is a *bytearray*.

Scripted Function Component

Like Connectors and Parsers, IBM Tivoli Directory Integrator allows you to fully program a Function Component using scripting. This is done by means of the template that the Scripted FC provides.

Note: The script for the Scripted Function Component is running in a separate JavaScript Engine. This means that the script cannot access any variables that are available, or have been set, in the normal hooks of an AssemblyLine.

Configuration

Configuration is relatively simple as all logic is in the Script pane.

Detailed Log
When checked, generates additional log messages.

Comment
Your own comments go here.

Using the FC

The bulk of the FC is in the script pane; in here, you must provide the logic that make up the FC.

To aid in programming, you are provided with stub functions as a reminder of the functions required to make a valid FC. These are:

initialize (fc,obj)
This function is called during the initialization phase of the AssemblyLine this FC is part of. The *obj* parameter is null when this method is called from an AssemblyLine FC.

terminate (fc)
This function is called during the termination phase of the AssemblyLine this FC is part of. Here is where you would release resources, etc.

perform (fc,obj)
This is the function that performs the actual work, and is called by the AssemblyLine at the point you positioned the FC. The *obj* parameter is the Entry containing your mapped out Attributes when this method is called from an AssemblyLine FC.

These correspond to the three main Function Interface methods. Each method is passed a Function parameter, which is this ScriptedFC.

Objects

Common objects (these are the same as for an AssemblyLine):

main The Config Instance (RS object) that is running.

task The AssemblyLine this Parser is a part of.

system
A `UserFunctions` object.

config The configuration for this element, that is, this Function Component. `config.getParent()` will be the FunctionConfig for the AssemblyLine FC, containing the Attribute Mapping and so on.

The following objects are the only ones accessible to the script Parser:

See also

“Script Connector” on page 255,
“Script Parser” on page 331,
"JavaScript Connector" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.

CBE Function Component

The CBE Function Component allows you to generate Common Base Event (CBE) event objects which can be written to CBE logs (which can be then viewed / managed using the Autonomic Computing Toolkit's Log and Trace analyzer) or issued to an IBM Common Event Infrastructure (CEI) server; alternatively, send it to an external application that is listening for CBE events.

Using the FC in Tivoli Directory Integrator, you must map your work entry attributes to the standard CBE attributes exposed in the Output Map of the CBE FC, so that when the AssemblyLine is run, the CBE FC creates a CBE Event object (and a CBE Event XML) and puts it back into the AssemblyLine's *work* entry.

You can also map a CBE object to the event attribute or XML representing a CBE object to the eventXml attribute of the OutputMap, so when the AL is run, the CBE FC retrieves all the standard CBE attributes and puts them back to the *work* entry.

Common Base Event (CBE)

Common Base Event (CBE) facilitates effective intercommunication among disparate enterprise components that support logging, management, problem determination, autonomic computing, and e-business functions.

An event encapsulates message data sent as the result of an occurrence of a situation. Events exchanged between and among applications in complex information technology systems allow these various facets of the system to interoperate, communicate and coordinate their activities. Fundamental aspects of enterprise management and e-business communications, such as performance monitoring, security and reliability, as well as fundamental portions of e-business communications, such as order tracking, are grounded in the viability and fidelity of these events.

The Common Base Event is defined as a new standard for enterprise management and business applications events. The Common Base Event definition ensures completeness of the data by providing properties to publish general information whenever a situation occurs. This general information provided by the Common Base Event is called the 3-tuple.

The following elements constitute the 3-tuple:

- The identification of the component that is reporting the situation
- The identification of the component that is affected by the situation (which may be the same as the component reporting the situation)
- The situation itself

The Common Event Infrastructure (CEI)

The Common Event Infrastructure (CEI) is IBM's implementation of a consistent, unified set of APIs and infrastructure for the creation, transmission, persistence and distribution of a wide range of business, system and network CBE formatted events. CEI is based upon the Autonomic Computing Division's CBE specification, which defines a standard format for event information, which devices and software use to keep track of transactions and other activity.

CEI is an embeddable technology intended to provide basic event management services to applications that require those services. This event infrastructure serves as an integration point for consolidation and persistence of raw events from multiple, heterogeneous sources, and distribution of those events to event consumers. Events are represented using the Common Base Event model, which is a standard defining a common representation of events that is intended for use by enterprise management and business applications. This standard, developed by the IBM Autonomic Computing Architecture Board, supports encoding of logging, tracing, management, and business events using a common XML-based format, making it possible to correlate different types of events that originate from different applications.

Input and Output attributes

This FC's Input and Output Attributes are identical to those of the “CBE Parser” on page 297.

Configuration

The CBE Function Component uses the following parameters:

Logger's Name
The name of the logger. This is an optional attribute and if you not define it defaults to LocalHostIP .

Mode This specifies whether this Function Component returns either CBE object from an Entry or Entry from a CBE object. Possible values are:

Entry -> CBE
This is the default mode that was used before Tivoli Directory Integrator 7.0. In this mode you must provide the required attributes of the OutputMap and receive the CBE object in the "event" attribute (also a XML representation in the "eventXml" attribute) of the InputMap.

CBE -> Entry
This mode that gives the user the ability to convert CBE event object to attributes. The CBE object is expected either in the *event* attribute of the OutputMap as plain Java object or in the *eventXml* attribute as XML representation. Then the attributes returned are provided in the InputMap. Specific for the XML is that the value of a tag (<situationType> someValue </situationType>) is taken as it is. This means that if it contains new lines or tabs then these characters will be returned to the attribute the same way.

Validate XML
Specifies whether to validate the XML against the XSD schema of the CBE specification. The default is "true".

Debug
Turns on debug messages. This parameter is globally defined for all Tivoli Directory Integrator components.

Generating a CBE Log XML

One of the primary needs for Tivoli Directory Integrator customers will be to have the ability to generate CBE compliant logs for their products so that other CBE log analyzers, like IBM's Autonomic Computing Toolkit's Log and Trace Analyzer (LTA), etc can be used to generate reports and analyze logs for different systems in a common and consistent manner.

You can create solutions which parse existing log files and generate new log files which are CBE compliant, or you can directly make your products communicate with Tivoli Directory Integrator, which will in turn generate logs in CBE compliant format.

Whatever the scenario, the CBE FC will have to be used to generate CBE events. You could accomplish this using the following steps:

1. Your put the required attributes to log inside the AssemblyLine's *work* entry. You may do this by parsing some existing product logs, or by reading a history database, etc.
2. The attributes are fed into the CBE FC by using the FC's *Output Map* operation, and the CBE FC generates an instance of a CBE Event object which has its various attributes set as per user passed values.
3. In one of the hooks (after event generation from CBE FC), a call to the **getCBELogXML( )** API (exposed in the CBE FC) can be made, and the newly created *event* object can be passed. The resulting

output string will be an XML fragment which adheres to the Hyades CBE Logging format. The string received from the **getCBELogXML( )** API can be (for example) set back into the work entry by calling the work.setAttribute( ) API.

```
var cbe = work.getObject("event");
var xmlString = com.ibm.di.fc.cbe.CBEGeneratorFC.getCBELogXml(cbe,false);
work.setAttribute("logXML",xmlString);
```

4. Then, using the File System connector with a LineReader parser, you can write this new attribute (containing the CBE Log XML) to any log file.

Emitting events to a CEI Server

With the aid of the CBE FC, you can emit/receive CBE events directly to the IBM CEI Server component. Currently, the IBM CEI server is a component that is shipped along with IBM WebSphere Process Server version 6.0. For an external Java application (not running inside WAS), the only way to emit events to a CEI server is to make use of the TEC web service available at: https://cs.opensource.ibm.com/projects/mainstream/.

This web service makes use of WS Notification to receive CBE events from external applications, and then making use of the IBM CEI SDK, transmits these CBE events to the CEI Server. This web service does not currently provide any means to consume or subscribe to events – and this is something that the TEC team may consider once WebSphere releases a standardized implementation of WS-notification.

The following steps illustrates how you can configure a solution to emit events to the IBM CEI server:

1. You put the required attributes inside the AssemblyLine's *work* entry. You may do this by parsing some existing product logs, or by reading a history database, etc.
2. The attributes are fed into the CBE FC by using the FC's Output Map operation, and the CBE FC generates an instance of a CBE Event object which has its various attributes set as per user passed values.
3. The event object is passed to Tivoli Directory Integrator's Axis web service FCs, which will serialize the CommonBaseEvent object; and send it over SOAP to the CEI webservice (on user defined port and WSDL address).
4. The CEI Web service will transmit this event to the CEI Server.

Function Component API

The CBE FC exposes the following methods (also see the Javadocs for this component):

public String convertCBEEventToXML (CommonBaseEvent event) throws Exception
This method will convert a CommonBaseEvent object to a XML string object. This XML will also be available by default in the eventXml attribute of the Input Map.

public String getCBELogXML (CommonBaseEvent event, boolean isCompleteXML)
This method is a wrapper over the org.eclipse.hyades.logging.java.CommonBaseEventLogRecord class's externalizeCanonicalXmlDocString( ) and externalizeCanonicalXmlString( ) API. This method can be used for obtaining a CBE Log XML. Whether the XML string returned is a complete XML document or just an XML fragment is decided by the isCompleteXML flag.

For more details see: http://archive.eclipse.org/tptp/4.2.0/javadoc/Platform/public/org/eclipse/hyades/logging/java/CommonBaseEventLogRecord.html

Also see “Generating a CBE Log XML” on page 396.

public static String mapCbeToEntry (CommonBaseEvent cbe, Entry entry)
This static method maps the fields of a Common Base Event object into the attributes of a Tivoli Directory Integrator Entry. The process is the reverse of what the CBE FC's 'perform' method does. All attributes in the resulting Entry are of type java.lang.String.

This method is accessible through Javascript in Tivoli Directory Integrator.

See also

- “CBE Parser” on page 297
- Autonomic Computing Toolkit (includes specification of the Common Base Event)
- Common Base Event best practices: Getting it right the first time. Highlights of the "Best Practices for the Common Base Event and Common Event Infrastructure" manual
- Common Base Event Best Practices Guide
- An example of generating Common Base Events with Tivoli Directory Integrator in examples/cbe_demo

SendEMail Function Component

The SendEMail Function Component uses the JavaMail API to send e-mails. By connecting to an Simple Mail Transfer Protocol (SMTP) server, the SendEMail Function Component can send e-mails to multiple recipients and can optionally attach multiple files to e-mails. You can also attach multiple files with different Multipurpose Internet Mail Extensions (MIME) types.

Note: Many Web-based e-mail services provide access only to browsers with HTTP. These services cannot be accessed using the SendEMail Function Component.

Schema

The SendEMail Function Component sends e-mails using an SMTP server; you can either use configuration parameters or map Attributes to operate this Function Component. The description of the input and output schemas is as follows:

Output Schema

attachments
A multivalued attribute. Each value specifies an attachment file to be added to the e-mail. Each value is either the absolute file path or a file path relative to the working directory. If the attribute is present it overrides the value of the attachments function component parameter.

In order to attach each file with different MIME type, you can provide the MIME type of attachment after the name of the file separated by ">".

For example:

```
SomeDocument.pdf>application/pdf
```

body String object that contains the body text of the mail. The Entry Attribute is required. An exception is thrown if the attribute is not present.

from The attribute specifies the content of the *from* field in the mail. If the attribute is present it overrides the value of the **From** function component parameter.

recipients
The attribute should be a comma separated list of the recipients of the mail. If the attribute is present it overrides the value of the **Recipients** function component parameter.

smtpServerHost
The attribute specifies the address of the SMTP server used to send the mails. If the attribute is present it overrides the value of the **SMTP Server Host** function component parameter.

smtpServerPort
The attribute specifies the port of the SMTP server used to send the mails. If the attribute is present it overrides the value of the **SMTP Server Port** function component parameter.

Subject
The attribute specifies the subject of the mail. If the attribute is present it overrides the value of the **Subject** function component parameter. A value for this field should be given, either mapped as an Attribute or provided in the **Subject** function component parameter, otherwise an exception is thrown.

replyTo
The attribute specifies the "reply-to" field of the message object. It contains a String object representing an array of mail addresses separated by commas. This String parameter is converted to InternetAddress Objects. Afterwards the created addresses are set to the outgoing message using the setReplyTo method of the message Object.

Input Schema

status This attribute is a java.lang.String object containing value "OK" if the mail has been sent successfully (that is, has been accepted by the SMTP Server).

Configuration

The SendEMail Function Component uses the following parameters:

SMTP Server Host
The parameter specifies the address of the SMTP server that sends mails. If this parameter is not set the `smtpServer` Entry Attribute should be mapped.

SMTP Server Port
The parameter specifies the port of the SMTP server that sends mails. If this parameter is not set the `smtpServerPort` Entry Attribute should be mapped. The mapped `smtpServerPort` Entry Attribute will take precedence over this parameter, even if it is set.

Username
This parameter is the user name used for SMTP authentication. Do not enter a value for this parameter if the SMTP Server does not require a user name and password authentication.

Password
This parameter is the password used for SMTP authentication.

Use SSL
Checking this parameter causes the FC to use Secure Sockets Layer (SSL) to communicate with the SMTP server.

From Specifies the content of the **From** field in the e-mail. If this parameter is not set, the `from` Entry Attribute should be mapped. The `from` parameter cannot contain spaces.

Recipients
This parameter is a comma separated list of the recipients' addresses. If it is not set, the `recipients` Entry Attribute should be mapped.

Subject
Specifies the subject of the e-mail.

Attachments
This multivalued parameter allows you to attach any files(s) you want to include with your message. Each value is either the absolute file path or a file path relative to the working directory. To set different a MIME type for individual attached files, add the MIME attachment type after file name. The MIME type and file name must be separated by the character >. For example:

```
SomeDocument.pdf>application/pdf
```

MIME Content Type
This parameter allows you to set the MIME content type of the e-mail's body; `text/plain` is the default value.

MIME Charset
Specifies the MIME charset to use for encoded words and text parts. If left blank the default system charset is used. Supported encodings can be found at: http://java.sun.com/j2se/1.5/docs/guide/intl/encoding.doc.html.

Reply To
The parameter is a comma separated list of the "Reply To" addresses. If this parameter is not set the `replyTo` Entry Attribute is mapped. This parameter is optional.

Debug
Turns on debug messages. This parameter is globally defined for all Tivoli Directory Integrator components

Memory Queue Function Component

Often referred to as the MemQueue FC. The Memory Queue FC encapsulates the functionality of the Tivoli Directory Integrator Memory Buffer Pipe (as present in the API) and provides a GUI to configure it. The FC contains two parts: the raw FC and the Config Editor (GUI) component. The raw FC encapsulates the calls to the memory buffer pipe. The GUI provides a way for the user to configure the behavior of a memory buffer pipe. The FC either returns a reference to the Memory Buffer Pipe object or reads/writes to it.

There can be multiple readers and writers for the same queue. Every writer has to obtain a lock before adding data . The writer has to release the lock before a reader can access it.

Note: Direct usage of the Memory Queue FC is deprecated in this release. It is much easier and also recommended to use the "Memory Queue Connector" on page 209 or directly use "The system object" on page 527 to create a new pipe, add data to the pipe and put data into the pipe. APIs for this functionality have been exposed in the System Object.

Configuration

Instance name
Name of the Tivoli Directory Integrator instance on which to create the Memory Buffer Pipe. The current instance is assumed if this is blank (default).

Pipe name
Name of the Memory Buffer Pipe to be created in the selected Instance.

Percentage memory to use
This determines what percentage of memory can be utilized by the memory queue. The default is 50.

Watermark
This is the threshold at which objects are persisted to the System Store. Note that the **Page Size** determines when pages are actually written, so the Watermark should be a multiple of the Page Size.

Page Size
Number of entries in one page.

Database name
A JDBC URL of an external database to use, or blank (the default) for the default System Store.

Username
Login username to the database used.

Password
Login password to the database used.

Table name
Table to use for paging.

Detailed Log
Check for additional log messages.

Using the FC

"The system object" on page 527 has a method called *getFunction(string name)* that returns an initialized instance of the FC. The returned object can be used to perform calls as in:

Using a simple call:

```
MemBufferQ pipe = system.getFunction("ibmdi.MemQueueFC").perform(null);
```

Using the *Entry* call:

```
var inp = system.newEntry();
inp.setAttribute ("test", "this is a sample entry");
MemBufferQ pipe = system.getFunction("ibmdi.MemQueueFC").perform(inp);
```

The Memory Buffer Queue FC returns a reference to a Memory Buffer Pipe for a null Entry object, performs a read operation on the Memory Buffer Pipe for the empty Entry object and a write operation on the Memory Buffer Queue for a non-empty Entry object.

The returned MemBufferQ object has two methods that can aid in managing the object: `purgeQueue()` and `deletePipe()`.

See also

"Memory Queue Connector" on page 209

"System Queue Connector" on page 227

Axis Java To Soap Function Component

The Axis Java-to-Soap Function Component (FC) is part of the Tivoli Directory Integrator Web Services suite.

Note: Due to limitations of the Axis library used by this component only WSDL (http://www.w3.org/TR/wsdl) version 1.1 documents are supported. Furthermore, the supported message exchange protocol is SOAP 1.1.

This component can be used both on the web service client and on the web service server side. This component receives an Entry or a Java object and produces the SOAP request (when on the client) or response (when on the server) message. It will provide the whole SOAP message, as well as separately the SOAP Header and the SOAP Body to facilitate processing and customization.

The component supports both RPC and Document style.

Configuration

Parameters

WSDL URL
The URL of the WSDL document describing the service

SOAP Operation
The name of the SOAP operation as described in the WSDL file

Return XML as
This drop-down list specifies whether the result is returned as a String or a DOM Element.

Complex Types
This parameter is optional; if specified, it should be a list of fully qualified Java class names (including the package name), where the different elements (Java classes) of this list are separated by one or more of the following symbols: a comma, a semicolon, a space, a carriage return or a new line.

Mode A flag that indicates whether this FC should generate a SOAP **request** (when deployed on the client) or a SOAP **response** (when deployed on the server) message

Use Multi Refs
This parameter is taken into account only when RPC-style web services are used. When Document-style web services are used this parameter has no effect on the generated SOAP message.

If checked and the web service used is an RPC-style web service, the generated SOAP message will use multi-refs. If **not** checked and the web service used is an RPC-style web service, the generated SOAP message will not **use** multi-refs.

Operation Parameters
This parameter is a list of Attribute names, where different Attribute names are separated by one or more of the following symbols: a comma, a semicolon, a space, a carriage return or a new line.

Detailed Log
When checked, generates additional log messages.

Comment
Your own comments go here.

Function Component Input

Entry or *Java Object*. If anything else is passed, an Exception is thrown.

Function Component Output

An Entry object with 3 attributes – one for the whole SOAP message, one for the SOAP Header and one for the SOAP Body. The SOAP message, Body and Header will be either XML strings or DOM objects, as specified by the **Return XML as** parameter.

Using the FC

This Function Component (FC) serializes the Java representation of a SOAP message into the XML representation of that SOAP message.

- If this FC is passed (a) an Entry with a "*soapFault*" Attribute, whose value is an object of type `org.apache.axis.AxisFault`, or (b) a Java object of type `org.apache.axis.AxisFault`, then the FC generates a SOAP Fault message containing the information stored in the passed *AxisFault* object.
- If the value of the **Return XML as** FC parameter is **String** then the SOAP response message is stored in the "*xmlString*" Attribute, if an *Entry* was passed to the FC. However, If the value of the **Return XML as** FC parameter is **DOMElement** then the generated SOAP message is stored in the "*xmlDOMElement*" Attribute , if an *Entry* was passed to the FC. If a *Java Object* array (Object[]) was passed to this FC, then the return value of the FC is either a `java.lang.String` object (when the value of the **Return XML as** FC parameter is **String**) or an `org.w3c.dom.Element` object (when the value is **DOMElement**).
- If the value of the "*soapFault*" Attribute passed in is not of type `org.apache.axis.AxisFault`, then an Exception is thrown.
- Each item from the value of the **Operation Parameters** FC parameter is the name of an Attribute, which must be present in the Entry passed to the FC. If any of these Attributes is missing, an Exception is thrown.
- If this FC is passed a *Java Object* array (Object[]) then it passes the SOAP operation each Java Object from this array in the order in which the Objects are stored in the array. If this FC is passed an *Entry*, then the order and values of the parameters passed to the SOAP operation are determined by the value of the **Operation Parameters** FC parameter.
- The order of the items from the value of the **Operation Parameters** FC parameter determines the order in which the Attribute values are passed as parameters to the SOAP operation.
- This FC is capable of generating (a) Document style SOAP messages, (b) RPC style SOAP messages and (c) SOAP Fault messages. The style of the message generated is determined by the WSDL specified by the **WSDL URL** FC parameter.
- The FC is capable of generating SOAP messages encoded using both "literal" encoding and SOAP Section 5 encoding. The encoding of the message generated is determined by the WSDL specified by the **WSDL URL** FC parameter.
- The parameter **Use Multi Refs** can mean different things, but is applicable only for RPC-style messages; when Document-style web services are used this parameter has no effect on the generated SOAP message. If checked and the web service used is an RPC-style web service, the generated SOAP message will use multi-refs. If **not** checked and the web service used is an RPC-style web service, the generated SOAP message will **not** use multi-refs.

  Note: The presence of the **Use Multi Refs** parameter is a consequence of using the Axis library to implement this FC. Currently when the Axis JavaToSoap FC serializes an RPC-style message it uses XML hrefs/multi-refs in the generated SOAP, and this breaks the Axis C++ library. That is why an Axis JavaToSoap FC configuration parameter is present to allow you to switch hrefs/multi-refs on and off.
- This FC is capable of generating SOAP messages containing values of complex types which are defined in the <types> section of a WSDL document. In order to do that this FC requires that (1) the **Complex Types** FC parameter contains the names all Java classes that implement the complex types used as parameters to the SOAP operation and (2) these Java classes' class files are located in the Java class path of Tivoli Directory Integrator.
- If this FC was passed an *Entry* object, then the FC stores the generated SOAP message Header and SOAP message Body (apart from the entire generated SOAP message) as Attributes in the returned *Entry*. If the value of the **Return XML as** FC parameter is **String** then the SOAP Header and Body are

stored in the "*soapHeaderString*" and "*soapBodyString*" Attributes respectively as `java.lang.String` objects. If the value of the **Return XML as** FC parameter is **DOMElement** then the SOAP Header and Body are stored in the "*soapHeaderDOMElement*" and "*soapBodyDOMElement*" Attributes respectively as `org.w3c.dom.Element` objects.

Custom serializers/deserializers

Serialization is the process of converting a Java object to an XML element. Deserialization is the process of converting an XML element to a Java object.

Both AxisJavaToSoap and AxisSoapToJava Function Components provide methods for registering XML type to Java type mappings with custom serializers/deserializers (by default all complex types are serialized/deserialized by Axis' org.apache.axis.encoding.ser.BeanSerializer/ org.apache.axis.encoding.ser.BeanDeserializer).

```
/**
   * This method is analogous to the 'registerTypeMapping' method in org.apache.axis.client.Call.
   * It can be used for configuring serialization/deserialization of Java types, for which the
   * default serializer/deserializer (org.apache.axis.encoding.ser.BeanSerializer/
   * org.apache.axis.encoding.ser.BeanDeserializer) is not suitable.
   */
    public void registerTypeMapping(Class javaType,
                                    QName xmlType,
                                    SerializerFactory serializerFactory,
                                    DeserializerFactory deserializerFactory)
```

This method can be invoked on an FC in the "After Initialize" Prolog FC hook through JavaScript like this:

```
var myClass = java.lang.Class.forName("mypackage.MyClass");
var myQName = new javax.xml.namespace.QName("http://www.myserver.com", "MyClass");
var mySerializerFactory = new mypackage.MySerializerFactory();
var myDeserializerFactory = new mypackage.MyDeserializerFactory();

myFC.getFunction().registerTypeMapping(myClass, myQName, mySerializerFactory, myDeserializerFactory);
```

Serialization/deserialization problems

By default all complex types are serialized/deserialized by Axis' org.apache.axis.encoding.ser.BeanSerializer/org.apache.axis.encoding.ser.BeanDeserializer. These default serializers and deserializers are not appropriate in certain rare cases. If you face a serialization/ deserialization error, it is likely that you need to provide a custom serializer/deserializer.

Here is one of the known cases:

XML Schema *list* type

When the XML Schema of the WSDL document defines an element to be of the *list* type:

```
<s:simpleType name="MyListType">
 <s:list>
   <s:simpleType>
     <s:restriction base="s:string">
       <s:enumeration value="One" />
       <s:enumeration value="Two" />
       <s:enumeration value="Three" />
     </s:restriction>
   </s:simpleType>
 </s:list>
</s:simpleType>
```

... you will see an error like the following:

```
at org.apache.axis.encoding.ser.ArrayDeserializer.characters(ArrayDeserializer.java:502)
 at org.apache.axis.encoding.DeserializationContext.characters(DeserializationContext.java:966)
 at org.apache.axis.message.SAX2EventRecorder.replay(SAX2EventRecorder.java:177)
 at org.apache.axis.message.MessageElement.publishToHandler(MessageElement.java:1141)
```

```
at org.apache.axis.message.RPCElement.deserialize(RPCElement.java:236)
at org.apache.axis.message.RPCElement.getParams(RPCElement.java:384)
at org.apache.axis.client.Call.invoke(Call.java:2467)
at org.apache.axis.client.Call.invoke(Call.java:2366)
at org.apache.axis.client.Call.invoke(Call.java:1812)
at com.ibm.di.fc.webservice.AxisEasyInvokeSoapWS.perform(Unknown Source)
```

The cause of the problem is that `org.apache.axis.encoding.ser.ArrayDeserializer` is not appropriate for *xsd:list* types. You should use the `org.apache.axis.encoding.ser.SimpleListDeserializer` deserializer instead. You can fix the problem using a script like the following in the **After Initialize** hook of the AxisJavaToSoap/ AxisSoapToJava FC:

```
var javaType = java.lang.Class.forName("[Ljava.lang.String;");
var xmlType = new javax.xml.namespace.QName("http://www.example.com", "MyListType");;
var serializerFactory = new org.apache.axis.encoding.ser.SimpleListSerializerFactory(javaType, xmlType);
var deserializerFactory = new org.apache.axis.encoding.ser.SimpleListDeserializerFactory(javaType, xmlType);
thisConnector.getFunction().registerTypeMapping(javaType, xmlType, serializerFactory, deserializerFactory);
```

See also

"Axis Soap To Java Function Component" on page 415

WrapSoap Function Component

The WrapSoap Function Component (FC) is part of the Tivoli Directory Integrator Web Services suite.

Note: Due to limitations of the Axis library used by this component only WSDL (http://www.w3.org/TR/wsdl) version 1.1 documents are supported. Furthermore, the supported message exchange protocol is SOAP 1.1.

This component is used to generate a complete SOAP message given the SOAP Body and optionally a SOAP Header.

Such a component is useful when the user customizes the content of the SOAP Body or creates it completely on his own (using Castor binding for example). This component will accept the contents of the SOAP Body and the SOAP Header and attributes for the SOAP Envelope, Header and Body XML elements (usually namespace declarations) and will create the complete SOAP message.

This is actually a helper FC that will save the user from error-prone processing of string or DOM objects to wrap his SOAP data into a complete SOAP message.

Configuration

Parameters

Input the SOAP Body and Header as
This drop-down specifies whether the SOAP Body and SOAP Header input values will be passed as String (that is, `java.lang.String`) or as DOM objects (`org.w3c.dom.Node`).

Return the SOAP message as
This drop-down specifies whether the complete SOAP message should be returned as a String or as a DOM object.

Header and Body tags present
Specifies whether the SOAP Body passed in an Attribute contains the <Body> tag and whether the SOAP Header passed in an Attribute contains the <Header> tag.

Attributes to add to the SOAP Envelope
Specifies the XML attributes and their values to include in the SOAP Envelope XML element.

Namespace declarations to add to the SOAP Envelope
Specifies Namespace declarations to add to the SOAP Envelope.

Attributes to add to the SOAP Body
Specifies the XML attributes and their values to include in the SOAP Body XML element.

Namespace declarations to add to the SOAP Body
Specifies Namespace declarations to add to the SOAP Body.

Attributes to add to the SOAP Header
Specifies the XML attributes and their values to include in the SOAP Header XML element.

Namespace declarations to add to the SOAP Header
Specifies Namespace declarations to add to the SOAP Header.

Detailed Log
Check to generate additional log messages.

Comment
Your own comments go here.

Function Component Input

Entry object – it has one Attribute for the SOAP Header (optional) and one Attribute for the SOAP Body.

If anything else is passed an Exception is thrown.

Function Component Output

An *Entry* object that contains the complete SOAP message.

Using the FC

The type and format of the entries processed and returned by this FC are highly dependent on the specified parameters, as clarified below.

- If the **Input the SOAP Body and Header as** FC parameter is **String** then the SOAP Body is passed in the "*soapBodyString*" Attribute and the SOAP Header is passed in the "*soapHeaderString*" Attribute. If the **Input the SOAP Body and Header as** FC parameter is **DOMElement** then the SOAP Body is passed in the "*soapBodyDOMElement*" Attribute and the SOAP Header is passed in the "*soapHeaderDOMElement*" Attribute.
- If the **Return the SOAP message as** FC parameter is **String** then the complete SOAP message is returned in the "*xmlString*" Attribute; however if it is specified as **DOMElement** then the complete SOAP message is returned in the "*xmlDOMElement*" Attribute.
- Each of the **Add attributes to...** parameters expects a list of XML attributes to be added to the target SOAP message element (envelope, header or body) tag in the created SOAP message. Each attribute-value pair is separated from the other attribute-value pairs by one of the following symbols: a space, a comma, a semicolon, carriage return or a line feed. The attribute name in an attribute-value pair is separated from the attribute value by an equals sign "=".
- Each of the **Namespace declarations to add to...** parameters expects a list of XML namespace declarations to be added to the SOAP message element (envelope, header or body) tag in the created SOAP message. Each namespace prefix-value pair is separated from the other namespace prefix-value pairs by one of the following symbols: a space, a comma, a semicolon, carriage return or a line feed. The namespace prefix in a prefix-value pair is separated from the namespace value by an equals sign "=".

InvokeSoap WS Function Component

Introduction

The Axis InvokeSoapWS Function Component (FC) is part of the Tivoli Directory Integrator Web Services suite.

Note: Due to limitations of the Axis library used by this component only WSDL (http://www.w3.org/TR/wsdl) version 1.1 documents are supported. Furthermore, the supported message exchange protocol is SOAP 1.1.

It is used to perform a web service call, given the input message for the call. It has no built-in SOAP parsing functionality and can be used with the "Axis Soap To Java Function Component" on page 415 and "Axis Java To Soap Function Component" on page 405 to provide a complete web service solution.

The InvokeSoapWS Function Component requires a complete SOAP request message. When called with a SOAP message the Function Component invokes the remote web service operation with this message. The Function Component returns the SOAP response message. The Function Component, however, does not perform any XML-Java binding (that is, the SOAP response message is not parsed) – the Function Component only returns the SOAP response message.

Authentication

The InvokeSoapWS FC supports the HTTP basic authentication method. If username and password parameters are filled, then the "authorization" HTTP header field is set with the proper credentials (as specified in the HTTP specification for using HTTP basic authentication). Before sending the username and password, the FC encodes them. The encoding used is Base64 and is done internally by the InvokeSoapWS FC.

Configuration

Parameters

WSDL URL
The URL of the WSDL document describing the service. his parameter is required; otherwise an exception is thrown on initialization.

SOAP Operation
The name of the SOAP operation as described in the WSDL file. This parameter is required; otherwise an exception is thrown on initialization.

Provider URL
The URL of the web service provider; substitutes the value in the WSDL; this parameter is provided to allow for dynamic provider switching. If this parameter is **Empty** then the value from the WSDL is used.

Login username
The login username sent to the server, using HTTP Basic Authentication. If the server requires authorization it uses this value and the next (Login password) to authenticate the client. The encoding used is Base64 and is done internally by the InvokeSoapWS FC.

Login password
The login password sent to the server, using HTTP Basic Authentication. If the server requires authorization it uses this value and the previous (Login username) to authenticate the client.

Input the SOAP message as
This drop-down list specifies whether the SOAP request message will be passed to the FC as a string or as a DOM object. This is a required parameter.

Return the SOAP message as
This drop-down list specifies whether the SOAP response message should be returned as a string or as a DOM object. This is a required parameter. If the parameter is not specified or has an invalid value, an exception is thrown on initialization. Also, if the SOAP request message does not conform to the format specified by the this parameter, an error will occur. However, it is ignored when invoking one-way web service operations (see"One-way web service operation support" on page 413.)

Detailed Log
When checked, will generate additional log messages.

Comment
Your own comments go here.

Function Component Input

An *Entry*, a `java.lang.String` object, or an `org.w3c.dom.Element` object – contains the complete SOAP request message.

If anything else is passed, an Exception is thrown.

If an Entry is passed to the FC and if the value of the **Input the SOAP message as** FC parameter is **String** then the SOAP request message must be stored in the "*xmlString*" Attribute of that Entry. If an *Entry* is passed to the FC and if the value of the **Input the SOAP message as** FC parameter is **DOMElement** then the SOAP request message must be stored in the "*xmlDOMElement*" Attribute.

If a non-*Entry* object (either String or Element) is passed to the FC and if the value of the **Input the SOAP message as** FC parameter is **String** then the SOAP request message must be passed as a `java.lang.String` object. If a non-Entry object (either String or Element) is passed to the FC and if the value of the **Input the SOAP message as** FC parameter is **DOMElement** then the SOAP request message must be passed as an `org.w3c.dom.Element` object.

Function Component Output

An Entry object with 3 attributes – one for the whole SOAP message, one for the SOAP Header and one for the SOAP Body. The SOAP message, Body and Header will be either XML strings or DOM objects, as specified by the **Return the SOAP message as** parameter. Refer to "Using the FC", next.

Using the FC

This Function Component makes a web service call by sending a SOAP request message and receiving a SOAP response message.

- If an *Entry* was passed to the FC, then if the value of the *Return the SOAP message as* FC parameter is *String* then the SOAP response message is stored in the "*xmlString*" Attribute; however, If the value of the *Return the SOAP message as* FC parameter is *DOMElement* then the SOAP response message is stored in the "*xmlDOMElement*" Attribute.
- Additionally, if this FC was passed an *Entry* object, then the FC stores the SOAP response Header and SOAP response Body (apart from the entire SOAP response message) as Attributes in the returned *Entry*. If the value of the **Output the SOAP message** as FC parameter is **String** then the SOAP Header and Body are stored in the "*soapHeaderString*" and "*soapBodyString*" Attributes respectively as `java.lang.String` objects. If the value of the **Return the SOAP message as** FC parameter is **DOMElement** then the SOAP Header and Body are stored in the "*soapHeaderDOMElement*" and "*soapBodyDOMElement*" Attributes respectively as `org.w3c.dom.Element` objects.
- If a non-*Entry* object was passed to this FC, then the return value of the FC is either a `java.lang.String` object (when the value of the *Return the SOAP message as* FC parameter is *String*) or an `org.w3c.dom.Element` object (when the value is *DOMElement*).
- This FC is capable of sending and receiving SOAP messages encoded using both "literal" encoding and SOAP Section 5 encoding.

- This FC is capable of sending and receiving SOAP messages containing values of complex types which are defined in the <types> section of a WSDL document.
- This FC sets the "*soapAction*" HTTP Header for the SOAP request message to the value specified in the WSDL document (whose location is specified by the *WSDL URL* FC parameter) for the given SOAP operation (whose name is specified by the *SOAP Operation* FC parameter).
- This FC sends the SOAP request message over HTTP to the web service address specified in the "*WSDL URL*\" parameter. If the "*WSDL URL*" parameter is missing or empty, the web service address specified in the WSDL document (whose location is specified by the *WSDL URL* FC parameter) for the given SOAP operation (whose name is specified by the *SOAP Operation* FC parameter) is used .
- This FC provides Username and Password parameters. If these parameters are provided, then the FC sets the basic authorization header and sends it to the server. It encodes the supplied username and password using encoding method base64; this is done inside the InvokeSoapWS FC

One-way web service operation support

WSDL 1.1 has four transmission primitives that a web service endpoint can support:

One-way
The endpoint receives a message.

Request-response
The endpoint receives a message, and sends a correlated message.

Solicit-response
The endpoint sends a message, and receives a correlated message.

Notification
The endpoint sends a message.

WSDL refers to these transmission primitives as operations. (More information on the subject can be found on: http://www.w3.org/TR/wsdl#_porttypes.)

The InvokeSoapWS Function Component supports only **request-response** and **one-way** web service operations. During the initialization phase, the InvokeSoapWS FC reads the configured WSDL document and checks whether the specified SOAP operation is one-way. If the operation is not one-way, it is assumed to be request-response.

The following is a sample WSDL fragment, which describes a request-response operation:

```
<operation name="myRequestResponseOperation">
 <input message="myInputMessage"/>
 <output message="myOutputMessage"/>
</operation>
```

And the following sample WSDL fragment describes a one-way operation:

```
<operation name="myOneWayOperation">
 <input message="myInputMessage"/>
</operation>
```

Note: One-way web service operations do not involve a server response – the client sends a request message but the server is not supposed to reply back (not even with a fault message). That is why the InvokeSoapWS does not return a response when invoking a one-way SOAP operation: If the 'perform' method of the FC is passed an *Entry* argument (for example when the FC is executed as a part of an AssemblyLine), the FC returns an **empty Entry**. If the 'perform' method of the FC is passed a java.lang.Object (for example when the FC is executed by a script), the FC returns **null**.

See also

"Axis EasyInvoke Soap WS Function Component" on page 421

Axis Soap To Java Function Component

The Axis Soap-to-Java Function Component (FC) is part of the Tivoli Directory Integrator Web Services suite.

Note: Due to limitations of the Axis library used by this component only WSDL (http://www.w3.org/TR/wsdl) version 1.1 documents are supported. Furthermore, the supported message exchange protocol is SOAP 1.1.

This component can be used both on the web service client and on the web service server side. This FC uses Axis' mechanism for parsing SOAP response (when on the client) or SOAP request (when on the server) to Java objects - as a complementary component to the AxisJavaToSoap FC. It is given a SOAP response/request message and returns the parsed Java objects either as standalone Java object(s) or capsulated in an Entry object.

This component supports both RPC and Document style.

Configuration

Parameters

WSDL URL
The URL of the WSDL document describing the service

SOAP Operation
The name of the SOAP operation as described in the WSDL file

Input the SOAP message as
This drop-down list specifies whether the SOAP message is specified as a string or as a DOM object.

Complex Types
This parameter is optional; if specified, it should be a list of fully qualified Java class names (including the package name), where the different elements (Java classes) of this list are separated by one or more of the following symbols: a comma, a semicolon, a space, a carriage return or a new line.

Mode This required parameter takes either a value of **Request** or **Response**. The value of Specifies whether this FC will parse SOAP request or SOAP response messages.

Detailed Log
If checked, will generate additional log messages.

Comment
Your own comments go here.

Function Component Input

Entry or *Java Object* representing the complete SOAP message.

If anything else is passed, an Exception is thrown.

Function Component Output

An *Entry* or a *Java Object* containing the Java representation of the SOAP request/response.

Using the FC

This Function Component parses a SOAP message and turns it into a Java Object.

- If this FC is passed a SOAP Fault message to parse, this FC returns a Java object of type `org.apache.axis.AxisFault`.

- In case this FC returns an `org.apache.axis.AxisFault` object, the FC stores this object in the "*soapFault*" Attribute if an *Entry* is passed to the FC; and if a *java.lang.Object* was passed then this FC returns the `org.apache.axis.AxisFault` object.
- If the value of the **Input the SOAP message as** FC parameter is **String** then the SOAP message to parse is read from the "*xmlString*" Attribute as a `java.lang.String`, provided an *Entry* is passed to the FC. If the value of the **Input the SOAP message as** FC parameter is **DOMElement** then the SOAP message to parse is read from the "*xmlDOMElement*" Attribute as an `org.w3c.dom.Element` object, provided an Entry is passed to the FC.
- If a *Java Object* is passed to this FC, then the SOAP message to parse is assumed to be the value of the passed Java Object as either a `java.lang.String` object (when the value of the **Input the SOAP message as**" FC parameter is **String**) or as an `org.w3c.dom.Element` object (when the value of the **Input the SOAP message as** FC parameter is **DOMElement**).
- This FC is capable of parsing (a) Document style SOAP messages, (b) RPC style SOAP messages and (c) SOAP Fault messages.
- This FC is capable of parsing SOAP messages encoded using both "literal" encoding and SOAP Section 5 encoding.
- This FC is capable of parsing SOAP messages containing values of complex types which are defined in the <types> section of a WSDL document. In order to do that this FC requires that (1) the **Complex Types** FC parameter contains the names all Java classes that implement the complex types used in the SOAP message and (2) these Java classes' class files are located in the Java class path of Tivoli Directory Integrator.
- If an *Entry* is passed to this FC and the message parsed is not a SOAP Fault message, then this FC returns the output parameters in Entry Attributes, whose names match the names of the SOAP Operation output parameters.

See also

"Custom serializers/deserializers" on page 407
"Axis Java To Soap Function Component" on page 405

Axis2 WS Client Function Component

The Axis2 WS Client FC is a Web Service client which is used to invoke a running Service. The Function Component uses the Apache Axis2 library to send the request to and to receive the response from the Web Service.

Both WSDL 1.1 (http://www.w3.org/TR/wsdl/) and WSDL 2.0 (http://www.w3.org/TR/wsdl20/) documents are supported.

Both SOAP 1.1 and SOAP 1.2 protocols are supported. Only *literal* SOAP messages can be used, *encoded* SOAP messages are not supported. This is a limitation of the underlying Axis2 library (version 1.4.0.1).

Using the FC

The Axis2 WS Client FC invokes a configured Web Service's operation and returns the response of it using the HTTP transfer protocol. The Function Component expects its payload in a Tivoli Directory Integrator hierarchical attribute named after the input message element in the WSDL document. The Axis2 WS Client FC will return the response in another hierarchical attribute named after the output message element in the WSDL document.

In the case of an In-Only operation the response hierarchical object will not be created and an empty Entry will be returned.

Supported Message Exchange Patterns

The Axis2 Web Service Client FC supports the following message exchange patterns (for more information see "Axis2 Web Service Server Connector" on page 25):

In-Only
The Axis2 library requires the pattern to be `http://www.w3.org/ns/wsdl/in-only` in the WSDL file

In-Out
The Axis2 library requires the pattern to be `http://www.w3.org/ns/wsdl/in-out` in the WSDL file

Robust-In-Only
The Axis2 library requires the pattern to be `http://www.w3.org/ns/wsdl/robust-in-only` in the WSDL file

SOAP Headers

See "SOAP Headers" on page 28.

Schema

The Axis2 components (Axis2WSClientFC and Axis2WSServerConector) receive and send data in the form of attributes with hierarchical structure. These attributes can be created in a Script Component, for instance, but their structure depends on the WSDL document provided as a parameter to the component.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
targetNamespace="http://www.example.com/HelloService.wsdl" >
...
<binding name="Hello_Binding" type="tns:Hello_PortType">
<soap:binding style="rpc | document"
transport="http://schemas.xmlsoap.org/soap/http"/>
<operation ... >
<input>
                <soap:body use="literal|encoded"? namespace="uri"?>
            </input>
            <output>
                <soap:body use="literal|encoded"? namespace="uri"?>
            </output>
```

```
        </operation>
</binding>
...
</definitions>
```

If this document is in accordance with WSDL 1.1 standard, there are two options:

1. The **style** of the operation in the soap binding can be **rpc**.

   Then an additional element is used to wrap the data that will be placed in the soap body.

   If the message that will be wrapped is a *request*, this wrapper element is named identically to the operation name. It has a **namespace** with the same value as either the optional namespace attribute defined in the soap:body element of the binding or, if the first is not present, the **targetNamespace** of the wsdl definition.

   Otherwise, if the wrapper is for the *response* message, its name will be formed by the operation's name plus the word "Response" and the namespace will be formed the same way as above. Each message part (parameter) appears under the wrapper, represented by an accessor named identically to the corresponding parameter of the call. Parts are arranged in the same order as the parameters of the call.

   A WSDL snippet:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <definitions name="HelloService"
     targetNamespace="http://www.example.com/wsdl/HelloService.wsdl"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:tns="http://www.example.com/wsdl/HelloService.wsdl"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <message name="SayHelloRequest">
   <part name="firstName" type="xsd:string"/>
   </message>
   <message name="SayHelloResponse">
   <part name="greeting" type="xsd:string"/>
   </message>
   <portType name="Hello_PortType">
   <operation name="sayHello">
   <input message="tns:SayHelloRequest"/>
   <output message="tns:SayHelloResponse"/>
   </operation>
   </portType>
   <binding name="Hello_Binding" type="tns:Hello_PortType">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
   ...
   </binding>
   ...
   ```

   The script for creating the hierarchical attribute that will be passed to the web service (the *request*: *SayHelloRequest*) is:

   ```
   var wrapper = work.createElementNS("http://www.example.com/HelloService.wsdl" ,"ns:sayHello");
   var message = work.createElement("firstName");
   message.appendChild(work.createTextNode("My Text Value"));
   wrapper.appendChild(message);
   work.setAttribute(wrapper);
   ```

   And the SOAP request looks like:

   ```
   <soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" >
      <soap:Header/>
      <soap:Body>
         <ns:sayHello xmlns:ns="http://www.example.com/HelloService.wsdl">
            <firstName>My Text Value </firstName>
         </ns:sayHello>
      </soap:Body>
   </soap:Envelope>
   ```

If you are assembling the response of the web service (from the Axis2WSServerConnctor for instance), the script is:

```
var wrapper = work.createElementNS("http://www.example.com/wsdl/HelloService.wsdl", "ns:sayHelloResponse");
var message = work.createElement("greeting");
message.appendChild(work.createTextNode("Returned Value"));
wrapper.appendChild(message);
work.setAttribute(wrapper);
```

And the SOAP response looks like:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" >
   <soap:Header/>
   <soap:Body>
      <ns:sayHelloResponse  xmlns:ns="http://www.example.com/wsdl/HelloService.wsdl">
         <greeting>Returned Value </greeting>
      </ns:sayHelloResponse>
   </soap:Body>
</soap:Envelope>
```

2. The style of the operation can be **document**.

   Then there are no additional wrappers, and the message parts appear directly under the SOAP Body element. We only need to create an attribute with the same name as the message part name, and its child element that hold the data we need to pass (in this example a string).

   For the same WSDL snippet:

```
...
<message name="SayHelloRequest">
<part name="firstName" type="xsd:string"/>
</message>
<message name="SayHelloResponse">
<part name="greeting" type="xsd:string"/>
</message>
<portType name="Hello_PortType">
<operation name="sayHello">
<input message="tns:SayHelloRequest"/>
<output message="tns:SayHelloResponse"/>
</operation>
</portType>
<binding name="Hello_Binding" type="tns:Hello_PortType">
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
...
</binding>
...
```

   The script for creating the request attribute looks like:

```
var message = work.createElement("firstName");
message.appendChild(work.createTextNode("My Text Value"));
work.setAttribute(message);
```

   And the SOAP request:

```
<soapenv:Envelope xmlns:soapenv=" http://www.w3.org/2003/05/soap-envelope/">
   <soapenv:Header/>
   <soapenv:Body>
      <firstName>My Text Value </firstName>
   </soapenv:Body>
</soapenv:Envelope>
```

   If you are assembling the response of the web service (from the Axis2WSServerConnctor for instance), the script should look like:

```
var message = work.createElement("greeting");
message.appendChild(work.createTextNode("Returned Value"));
work.setAttribute(message);
```

   And the SOAP response:

```
<soapenv:Envelope xmlns:soapenv=" http://www.w3.org/2003/05/soap-envelope/">
   <soapenv:Header/>
   <soapenv:Body>
      <greeting>Returned Value </greeting>
   </soapenv:Body>
</soapenv:Envelope>
```

For additional information on this subject, see http://www.w3.org/TR/wsdl/, section 3.5.

If the supplied WSDL file is in accordance with WSDL 2.0 standard, then:

The situation is similar to the document operation style - there is no need for additional wrappers.

The hierarchy of the attribute must comply with the structure of the message as defined by the *types* block of the WSDL file.

For SOAP Headers information see “SOAP Headers” on page 28.

The Axis2WSClientFC has the ability to log the HTTP headers of received SOAP responses when in detailed log mode. That way you can easily access this information. In addition, HTTP attributes can be mapped in the Input Map of the FC, thus setting specific headers of the SOAP request. . For more information on HTTP Headers, see “HTTP Server Connector” on page 115 and “Axis2 Web Service Server Connector” on page 25.

Configuration

The Axis2 WS Client Function Component has the following parameters:

WSDL URL
The location of the WSDL file which will be used for the WebService invocation.

Service
The name of the Service (as written in the WSDL file) which is to be invoked. This parameter has a button with an assigned script to it; this script will display a drop down box from which you can select the desired Service name. However, for the script to function you must first specify a WSDL file because it shows the Services in it.

Operation
The name of the SOAP Operation which is to be invoked. This parameter has a button with an assigned script to it; this script will display a drop down box from which you can select the desired operation. However, for the script to function you must first select a Service because the intention is to show the operations associated with the service.

Endpoint
The name of the Endpoint (as written in the WSDL file) which corresponds to the WebService to be invoked. This parameter has button with assigned script to it; this script will display a drop down box from which you can select the desired Endpoint name. However, for the script to function you must first specify as Service because it shows the endpoints associated with it.

Username
The username to be used for HTTP Basic Authentication invocations.

Password
The password to be used for HTTP Basic Authentication invocations.

When configuring this Connector, you first configure the WSDL file, after which you select the Service. Next, you select the SOAP Operation and Endpoint.

See also

The example in *TDI_install_dir*/examples/axis2_web_services.

Axis EasyInvoke Soap WS Function Component

The Axis EasyInvokeSoapWS Function Component (FC) is part of the Tivoli Directory Integrator Web Services suite.

Note: Due to limitations of the Axis library used by this component only WSDL (http://www.w3.org/TR/wsdl) version 1.1 documents are supported. Furthermore, the supported message exchange protocol is SOAP 1.1.

This is a "simplified" web service invocation component: it is a stand-alone FC with its own Config screen, but internally performs the same functions as the following three FCs: AxisJavaToSoap, InvokeSoapWS and AxisSoapToJava.

The functionality provided is the same as if you chain and configure these three FCs in an AssemblyLine. When using this FC you lose the possibility to hook custom processing, that is, you are tied to the processing and binding provided by Axis, but you gain simplicity of setup and use.

Note that some scenarios cannot be handled by this Function Component, and you should use the aforementioned combination of AxisJavaToSoap, InvokeSoapWS and AxisSoapToJava instead. Such scenarios are:

- Need to process SOAP headers
- Serialization/deserialization problems (see “Serialization/deserialization problems” on page 407 for more information on serialization/deserialization problems and how to register custom serializers/deserializers to handle these problems)

Configuration

Parameters

WSDL URL
The URL of the WSDL document describing the service. This parameter is required; otherwise an exception is thrown on initialization.

SOAP Operation
The name of the SOAP operation as described in the WSDL file. This parameter is required; otherwise an exception is thrown on initialization.

Login username
The login username sent to the server, using HTTP Basic authentication. If the server requires authorization it uses this value and the next (Login password) to authenticate the client. The encoding used is Base64 and is done internally by the InvokeSoapWS FC.

Login password
The login password sent to the server. If the server requires authorization it uses this value and the previous (Login username) to authenticate the client.

Complex Types
This parameter is optional; if specified, it should be a list of fully qualified Java class names (including the package name), where the different elements (Java classes) of this list are separated by one or more of the following symbols: a comma, a semicolon, a space, a carriage return or a new line.

Operation Parameters
The list of ordered SOAP operation parameter names. This parameter is required if the SOAP operation has any parameters and the FC is passed an *Entry*; if the SOAP operation has no parameters and the FC is passed an *Entry* then this parameter must be empty. If no *Entry* is passed, the content of the parameter is not relevant.

- If specified, the parameter must contain a list of Attribute names, where different Attribute names are separated by one or more of the following symbols: a comma, a semicolon, a space, a carriage return or a new line.
- Each item from this list is a name of an Attribute, which must be present in the *Entry* passed to the FC. If one of these Attributes is missing, an Exception is thrown.
- The order of the items from the list determines the order in which the Attribute values are passed as parameters to the SOAP operation.

Timout
The time for retrieving the data (in seconds, with the possibility to specify a decimal fraction in milliseconds. If smaller than 0.001 or explicitly set to 0 then will wait forever). Default value is 60.

Detailed Log
Checking this box causes additional log messages to be generated.

Comment
Your own comments go here.

Security and Authentication
The AxisEasyInvokeSoapWS FC uses HTTP basic authentication. When the username and password parameters of the FC are provided, then this information is sent to the server; if it requires authentication it takes these two parameters for username and password.

Function Component Input
Entry or a *Java object* representing the web service input data. If anything else is passed, an Exception is thrown.

Function Component Output
Entry or a *Java object* representing the web service output data.

Using the FC
This Function Component (FC) provides a relatively simple way of invoking SOAP over HTTP web services.

This is how the communication flows:

Web service client <–> AxisEasyInvokeSoapWS FC <–> org.apache.axis.client.Call <–> Web service

The following usability notes apply:
- If this FC is passed a *Java Object* array (*Object[]*) then it passes to the SOAP operation each Java Object from this array in the order in which the Objects are stored in the array. If this FC is passed an *Entry*, then the order and values of the parameters passed to the SOAP operation are determined by the value of the SOAP Operation FC parameter.
- This FC is capable of generating and parsing Document-style SOAP messages and RPC-style SOAP messages as well as parsing SOAP Fault messages. The style of the message generated is determined by the WSDL specified by the WSDL URL FC parameter.
- The FC is capable of generating and parsing SOAP messages encoded using both "literal" encoding and SOAP Section 5 encoding. The encoding of the message generated is determined by the WSDL specified by the WSDL URL FC parameter.
- This FC is capable of generating and parsing SOAP messages containing values of complex types which are defined in the <types> section of a WSDL document. In order to do that this FC requires that (1) the Complex Types FC parameter contains the names all Java classes that implement the complex types used as parameters to the SOAP operation and (2) these Java classes' class files are located in the Java class path of Tivoli Directory Integrator.

- If an *Entry* is passed to this FC and the SOAP response message returned by the server is not a SOAP Fault message and there is a single output parameter of the SOAP Operation, then this FC returns the parameter in the "*return*" Attribute. (Due to Axis 1.1 specifics, if a SOAP operation has a single output parameter, this parameter is considered the return value of the operation. And if a SOAP operation has several output parameters, its return type is considered to be void.)
- If an *Entry* is passed to this FC and the SOAP response message returned by the server is not a SOAP Fault message and there are several output parameters of the SOAP Operation, then this FC returns the output parameters in *Entry* Attributes, whose names match the names of the SOAP Operation output parameters.
- If an *Object[]* with the input parameters of the SOAP operation is passed to this FC and the SOAP response message returned by the server is not a SOAP Fault message, the result is of type *Object[]*, where the first element is the return value of the SOAP operation (null if the operation is void) and the rest are the output parameters of the operation.

  Object[] –> AxisEasyInvokeSoapWS FC –> Object[]

  or

  Entry –> AxisEasyInvokeSoapWS FC –> Entry
- This FC provides username and password parameters. If these parameters are specified, then the FC sets basic authorization header and sends it to the server. It encodes the supplied username and password. The used encoding method is base64 and is done inside the InvokeSoapWS FC.
- The timeout field specifies the time the invocation will wait for response. On a timeout an exception will be thrown notifying the user that the webservice did not respond in a timely fashion. The default timeout used by Axis 1.4 is 60 seconds. The timeOut parameter can accept a double value formatted to the third digit after the point and is in seconds.

See also

"InvokeSoap WS Function Component" on page 411

Complex Types Generator Function Component

The Complex Types Generator Function Component is part of the Tivoli Directory Integrator Web Services suite.

Note: Due to limitations of the Axis library used by this component only WSDL (http://www.w3.org/TR/wsdl) version 1.1 documents are supported. Furthermore, the supported message exchange protocol is SOAP 1.1.

This Function Component is used for generating a JAR file, which contains the Java class files implementing the complex data types defined in a schema either internal to or referenced by a WSDL. This JAR file can then be used by the other Web Service FCs in order to serialize and parse SOAP messages containing these complex data types.

Please note that this FC is not supposed to be "run" as part of an AssemblyLine for example. Here is the way this FC is supposed to be used:

1. Place it in an AssemblyLine
2. Fill in its parameters
3. Click the **Generate complex types** button to create the JAR file.

After the desired JAR file has been created the FC can be either disabled or deleted altogether from the AssemblyLine – the FC does not provide any runtime functionality whatsoever.

Configuration

Parameters

WSDL URL
The value of this parameter must be either a valid URL string or a file system path (either absolute or relative) specifying the location of a WSDL file.

WSDL2Java Options
The value of this parameter is a command-line-like list of options for the Axis WSDL2Java utility. The FC passes this list of options to the WSDL2Java utility when generating Java source files from WSDL. These options can be used to alter the default behavior of the WSDL2Java utility.

JAR file name
The value of this parameter must be the name of the JAR file (either absolute or relative) to be created.

JDK Path
The path to a Java Development Kit installation. If left empty the utility assumes that the Java compiler "`javac`" and the "`jar`" tools are on the system executable path.

Note: The implementation of this FC requires a minimum version 1.4 of the JDK.

Generate Java Source Files
If this box is checked (which is the default), then the FC utility generates Java source files from the specified WSDL. If unchecked, then the FC utility skips the generation of Java source files and only performs the compilation and the JAR creation. Setting this parameter to false (that is, unchecked) is useful when you want to write the implementation of the complex types yourself or you want to modify auto-generated Java source files (setting this parameter to true will overwrite any manually edited/written Java source files).

Function Component Input and Output

You run the FC JAR creation utility by pressing the "**Generate complex types**" button.

- The Java **source** files are output and then read from "<*installation_folder*>/temp/ComplexTypesJavaFiles". If this folder does not exist it is automatically created.
- The Java class files are output and then read from "<*installation_folder*>/temp/ComplexTypesClassFiles". If this folder does not exist it is automatically created.

Note: Before creating any output files (Java source or class files, the JAR file) the previously generated files are deleted.

Troubleshooting

If the ComplexTypesGenerator FC displays an error message box and you need further information about the error that has occurred do the following steps:

1. Change the log level of the *log4j.logger.com.ibm.di.admin* logger in "<*installation_directory*>/log4j.properties" to DEBUG. For example change the line *log4j.logger.com.ibm.di.admin=WARN* to *log4j.logger.com.ibm.di.admin=DEBUG*.
2. Restart the Config Editor.
3. Run the ComplexTypesGen utility again.

Delta Function Component

Introduction

The Delta Component provides lookup, add and delete access to the Delta services outside of the normal Connector modes. The Delta Function Component allows the Delta functionality to be placed anywhere in the AssemblyLine. This way entries read from the input source can be modified before computing the Delta changes and applying them to the Delta Store.

Configuration

The Delta Function Component has the following parameters:

Unique Attribute Name
The name of an attribute that holds a unique value in a given data source. Data sources with duplicate keys cannot be subjected to the delta function, except when **Allow duplicate Delta keys** is enabled.

Delta Store
The table in the System Store that holds the Delta information from previous runs for this Connector, so as to be able to detect differences on subsequent runs. When this parameter is empty a default name composed from the "AssemblyLines" literal string, the AssemblyLine name and the component name is used (for example, "`AssemblyLines_AL1_DeltaFunc`").

Read Deleted
If checked, the AssemblyLine will inject deleted entries into the AssemblyLine run when the Iterator has completed iterating, that is, finished input. The operation code will indicate that this Entry was deleted in the input source. Note that delete-tagged Entries are not removed from the Delta Store unless you also enable the Remove Deleted flag.

Remove Deleted
If checked, the deleted entries from the input source are deleted from the Delta Store, such that they will not be detected again in subsequent runs.

Return Unchanged
If checked, any unchanged entries in this run are injected into the AssemblyLine.

Commit
Selects when to commit changes to the Delta Store as a result of iterating through the input. Choices are:

- After every database operation
- On end of AL Cycle
- On Connector close
- No autocommit

The default is **After every database operation**.

Row Locking
Selects the transaction isolation level for the connection to the Delta Store. For more information refer to subsection "Row Locking" in section "Delta feature for Iterator mode" in *IBM Tivoli Directory Integrator V7.1 Users Guide*. Possible values are:

- READ_UNCOMMITTED
- READ_COMMITTED
- REPEATABLE_READ
- SERIALIZABLE

The default is **READ_COMMITTED**.

Faster algorithm
When checked, instructs the AssemblyLine to use a faster algorithm to compute changes, at the expense of more memory use. In essence, it does not write unchanged entries to the Delta store; instead it remembers keys in memory if **Read Deleted** is set to *true*.

Allow duplicate Delta keys
When the Delta feature is enabled for Changelog/Change Detection Connector in long running AssemblyLines, an Entry can be modified more than once. These modifications will result in receiving the Entry a second time and this will cause the Duplicate delta key exception to be thrown. Checking this parameter allows Entries with duplicate key attributes (specified in the **Unique Attribute Name** parameter) to be processed by Iterator Connectors with enabled Delta.

Attribute List
A comma-separated list of attributes whose changes will be detected or ignored during the compute changes process. The changes in listed attributes will be affected by the **Change Detection Mode** parameter that specifies whether to ignore or detect them. For more information about this parameter see subsection "Detect or ignore changes only in specific attributes" in section "Delta feature for Iterator mode" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.

Change Detection Mode
Specifies changes in which attributes will be detected or ignored. For more information about this parameter see subsection "Detect or ignore changes only in specific attributes" in section "Delta feature for Iterator mode" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.

Using the Function Component

The Delta Function Component provides the same functionality as an enabled Delta tab for connectors in Iterator mode; see section "Delta" in *IBM Tivoli Directory Integrator V7.1 Users Guide*. This function component allows Delta detection and applying logic to be placed anywhere in the AssemblyLine.

Since the Delta FC is a function component it requires an Entry on which to perform its Delta function. Therefore when the input source reaches end of data, the Delta component will have nothing to process. So when the **Read Deleted** parameter is selected the deleted entries will be returned by the Delta Function Component only if you feed the component with some dummy empty entries. These entries will signal the function component to start returning any deleted entries.

Similar to an Iterator connector with Delta enabled, the Delta Function Component displays delta statistic at the end of Assembly Line execution (for example, `"Add:3, Modify:1, CallReply:5, Skip:2, Nochange:2"`).

Example

The following example demonstrates how to synchronize an input source with the TDI Delta Store using the Delta FC when the **Read Deleted** parameter is enabled. The deleted entries from the input source will be marked with a *delete* delta operation and returned to the AssemblyLine.

Steps to execute:

1. Add connector(s) in Iterator mode that will iterate over the input data source.
2. Add a Script Connector as the last connector in the Feed section.
3. Add this code in its getNextEntry() method:

```
r = task.getResult();

if (r != null){
 if (r.size() > 0){
  entry = system.newEntry();
  result.setStatus(1);        // OK
```

```
} else {
 result.setStatus(0);        // end of input
}
    }
```

4. Add some custom logic or components that modify the work entry.
5. Add Delta Function Component and configure it to iterate over deleted entries.
6. Start the AssemblyLine.
7. Delete some entries in the input data source.
8. Start the AssemblyLine again and receive the deleted entries tagged as *delete*.

Note: At the end the DeltaFC will return a dummy empty entry. Checking this entry can be used by the Script Connector's getNextEntry() method to determine when to stop returning dummy entries. An empty entry is also returned when there are no deleted entries and **Read Deleted** is enabled.

Remote Command Line Function Component

The Remote Command Line Function Component (Remote CLFC) enables command line system calls to be executed on remote machines. The design and implementation uses the RXA toolkit v2.2 to connect to remote machines, execute the commands and return the results. The returned output can then be parsed to be consumed one value at a time and detect any problems with the executed command.

The Remote CLFC has the ability to connect to remote machines using any of the following protocols: RSH, REXEC, SSH, AS400 or Windows. You can select which of the protocols will be used; however, if left to the default value of 'ANY', the FC will attempt to connect to the remote machine using each of the available protocols one-by-one until a successful connection is made.

The RXA libraries used by this FC support interactive SSH sessions only; non-interactive SSH sessions are **not** supported.

You will need to provide information about the remote machine including hostname, username and password. If the connection is being made using the SSH protocol then you have the option of providing a keystore name and passphrase instead of using a password for authentication.

Note: SSH Connections are typically associated with Linux/UNIX and z/OS hosts. However, by installing Cygwin and the Cygwin *openssh* package on the Windows target machine the SSH protocol can be used with those targets as well. In addition, z/OS targets can be reached using the SSH protocol also.

Configuration

Target Machine Hostname
The hostname (address) of the target machine. This is a required parameter.

Remote User
The name of a user with Administrative privileges on the target machine.

Password
The password for the user (specified as **Remote User**) on the target machine. This parameter may be optional in the case of SSH connections using a keystore, as well as for RSH connections.

Keystore Path
Full path to the file containing the keystore. This parameter is optional, and only used for SSH connections.

Passphrase
The passphrase that protects your private key, in the keystore specified by the **Keystore Path** parameter above.

Connection Protocol
Select from 'ANY', 'SSH', 'RSH', 'REXEC', 'AS400' and 'WIN'. This designates what protocol to use when connecting to the remote machine. See “Using the FC” on page 433 for more details.

Port The port to use to connect to the target machine.

Command
The command that is to be executed on the target machine. This is overridden if an output attribute 'command.line' has been provided. This is a required parameter, unless the command.line Attribute is supplied in the Output map.

Stdin source file (local)
The path to the file on the local system that is to be used as standard input to the command specified. This parameter is optional.

Stdin destination directory (remote)
The path to an existing destination directory on the target where you want the standard input

file, designated by **Stdin source file (local)**, to be copied. If a value for **Stdin source file (local)** has been provided, but no value for the destination then a random temporary directory will be created on the remote machine. Note that the file is copied temporarily; once the command has finished execution, the copy on the remote machine is deleted.

Convert Stdin source file to character set of target system
If checked, the Stdin source file will be converted to the character set of the target system; otherwise, the current encoding of file will be maintained.

Timeout (ms)
The desired CPU timeout period in milliseconds. If the operation does not complete within the specified duration then the operation is cancelled. This parameter is optional. An unspecified or 0 (zero) value indicates Unlimited, that is, no computational time limit.

Note: The timeout is a measure of the CPU clock time of the Remote CLFC process, not a measure of the actual time elapsed since process initiation. Commands that are not computationally intensive will not timeout in the specified time if they have not reached their computational time limit.

Initial connection timeout (ms)
An optional Remote CLFC parameter that defines a timeout period for the initial connection to the target system. This has no effect on AS400 targets.

Enable SSL for AS400?
This parameter governs whether an SSL connection is enforced on the AS400 (i5/OS) connection. If checked an SSL connection will be attempted to the AS400 target (SSL must be installed and configured on the AS400 target system). The default is unchecked.

AS400 Proxy Server Name
This parameter defines an AS400 proxy server if so required.

Run AS400 Program?
An optional Remote CLFC parameter that defines the type of command execution to use for an AS400 (i5/OS) connection. AS400 programs have the extension .PGM. Arguments for these AS400 Programs can be specified using the Entry Attribute **command.args**.

The default value is unchecked.

Enable RXA Internal Logging?
Enabling this will allow the RXA internal logger to generate log messages in the AssemblyLine log file.

AS400 Command Line Arguments Character Encoding
The character encoding to use for AS400 command line arguments. The default character encoding of the JVM is used if not specified. This configuration parameter is optional and only applies when the **Run AS400 Program?** parameter is set. Failure to set this value to the proper encoding of the target AS400 box can cause the command line parameter strings to be corrupted if the encoding of the JVM on the remote machine does not match the default encoding of the AS400 machine where the AS400 Program will be run.

Detailed Log
Enabling this will generate debug log messages.

Function Component Input

Some of the parameters configured in the Configuration screen of the Remote CLFC can be provided as Attributes mapped from the work Entry in the Input Map. When present and non-empty, they take precedence over the parameters in the Configuration screen:

command.line
The command that is to be executed on the target machine. This attribute has to be defined in the Output map, and will replace the "**Command**" parameter defined under the Config tab.

command.args
A multi-valued Attribute whose values each are command line arguments. Required when executing AS400 Programs.

command.args.delim
This Attribute specifies the command/Program argument delimiter. If not specified the default is a single white space character.

stdin.source
This attribute, of type java.io.String, represents the path to the file on the local machine that is to be used as standard input for the specified command.

stdin.destination
This attribute, of type java.io.String, represents the path where the transferred file should be stored on the remote machine.

In other words, if an attribute called *command.line* is provided in the input entry object then any command that was entered in the Config Editor will be disregarded. This allows you to call the Remote CLFC repeatedly by other components in the AssemblyLine to perform different commands.

Function Component Output

Once the Remote CLFC has executed the command specified by either the *command.line* attribute or the **Command** configuration parameter as discussed above, the FC makes the following attributes available for attribute mapping:

command.returnCode (int)
The return code that resulted from executing the remote command.

command.error (String)
The standard error message, if any, that was generated when the command was run.

command.out (String)
The standard output message, if any, that was generated when the command was run.

Using the FC

The Remote CLFC may be used within an AssemblyLine containing other Tivoli Directory Integrator components such as Connectors and other Function Components. To function correctly, you must configure the Remote CLFC correctly using the Config Editor. When it is initialized it will establish a connection with the remote machine and then when its `perform()` method is called (normally when it is reached in the AssemblyLine it is part of), it will execute its command on the target.

Upon completion, the `perform()` method will return an Entry object containing the three output attributes described above: *command.returnCode, command.error* and *command.out*. These attributes will then be available to other Tivoli Directory Integrator components further down in the AssemblyLine.

If you use the FC to perform a command that returns a list of messages in Standard Out such as a directory listing then the Remote CLFC would need to be used in conjunction with other Tivoli Directory Integrator components, like a Parser, in order to extract the individual entries from the *command.out* String object and process them one at a time.

Configuring the Target System

The target machines must satisfy the following requirements:

Windows Targets
Using the **WIN** protocol: Windows XP targets must have Simple File Sharing disabled for Remote Execution and Access to work. Simple Networking forces all logins to authenticate as "guest". A guest login does not have the authorizations necessary for Remote Execution and Access to function.

To disable Simple File Sharing, you need to start Windows Explorer and click **Tools->Folder Options**. Select the **View** tab, scroll through the list of settings until you find **Use Simple File Sharing**. Remove the check mark next to **Use Simple File Sharing**, then click **Apply** and **OK**.

Windows XP includes a built-in firewall called the Internet Connection Firewall (ICF). By default, ICF is disabled on Windows XP systems, except on Windows XP Service Pack 2 where it is on by default. If either firewall is enabled on a Windows XP target, it will block attempted accesses by Remote Execution and Access. On Service Pack 2, you can select the File and Printer Sharing box in the Exceptions tab of the Windows Firewall configuration to allow access.

The target machine must have remote registry administration enabled (which is the default configuration) in order for Remote Execution and Access to run commands and execute scripts on the target machine.

The default hidden administrative disk shares (such as C$, D$, etc) are required for proper operation of Remote Execution and Access.

Cygwin Targets

Using the **SSH** protocol: To use SSH logins to remote Windows computers, you must download Cygwin from http://cygwin.com and install it on each Windows machines that your application will target. Complete documentation for Cygwin is available at http://cygwin.com.

To use Remote Execution and Access applications on Cygwin targets, you will need to install up to two additional Cygwin packages that are not part of the default Cygwin installation. From http://cygwin.com, download and install openssh, which is in the *net* category of Cygwin packages. openssh contains the ssh daemon that is needed to support SSH logins on Cygwin targets. Another package, cygrunsrv, which is in the *admin* category of packages, provides the ability to run the ssh daemon as a Windows service. If you do not wish to run the ssh daemon as a service, this package is optional.

MKS Targets

The MKS toolkit is an alternative to using Cygwin on windows machines. For more information refer to http://www.mkssoftware.com/. To use MKS from the windows command line, add the path to *MKS_ Installation*`/bin` to the PATH environment variable. By default in MKS, SSH is configured to use Username password authentication. To set up passwordless authentication, a pair of public and private keys must be generated using the ssh-keygen utility (available with the MKS Toolkit and on most UNIX systems) and copy them to the machine to which to connect.

If connecting to a secure shell service or daemon that is derived from the OpenSSH version of secure shell (such as the secure shell service (sshd) from MKS Toolkit), the protocol version 1 RSA keys must be appended to the hosts's `~/.ssh/authorized_keys` file and protocol version 2 RSA and DSA keys to the host's `~/.ssh/authorized_keys2` file where `~/` is the home directory of the account on the remote host.

UNIX and Linux Targets

Using **SSH**, **RSH** or **REXEC** protocols: The RXA toolkit this FC uses does not supply SSH code for UNIX machines. You must ensure SSH is installed and enabled on any target you want to access using SSH protocol. OpenSSH 3.71, or higher, contains security enhancements not available in earlier releases.

RXA cannot establish connections with any UNIX target that has all remote access protocols (rsh, rexec, or ssh) disabled.

In all UNIX environments except Solaris, the Bourne shell (sh) is used as the target shell in UNIX environments. On Solaris targets, the Korn shell (ksh) is used instead due to problems encountered with sh.

In order for RXA to communicate with Linux and other SSH targets using password authentication, you must edit the file /etc/ssh/sshd_config file on target machines and set:

```
PasswordAuthentication yes (the default is 'no')
```

After changing this setting, stop and restart the SSH daemon using the following commands:

```
/etc/init.d/sshd stop
/etc/init.d/sshd start
```

When using the rsh / rexec connection protocol, the Tivoli Directory Integrator Server must be running as a privileged user (root on Unix or a user with Administrator privileges on Windows). The connection will fail if this requirement is not met. The rsh and rexec connection protocols require that the source port be "trusted" (port number less than 1024), however these platforms restrict creation of trusted port connections to privileged users.

For further details on how to configure SSH between the local machine and the target, either using password authentication or a keystore, please refer to the relevant OpenSSH documentation at http://www.openssh.com.

z/OS Targets

Using SSH and z/OS, VMS shell commands can be executed on mainframe zSeries systems. Commands executed using this type of connection are very Unix-like as they are run in the simulated Unix shell environment.

Note: The target Tivoli Directory Integrator Server you run this on should be in ASCII mode as required by the underlying RXA libraries. Refer to "ASCII Mode" in *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide* for more information.

AS400 Targets

AS400 targets require the IBM Toolbox for Java (V5.2 recommended) to be installed along with a suitable JRE. The IBM Toolbox for Java is also required on the Tivoli Directory Integrator server where the JAR files will be placed in the *TDI_install_dir*/jars/3rdParty/IBM directory. The commands and programs themselves have to be located under the QSYS library on the iSeries system.

When enabling the AS400 SSL connection option, additional configuration is required for self signed certificates. In this case, the signing certificate must be added to the Java Security CA Certificate store (*jre_directory*/lib/security/cacerts). Further information can be found here: http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/keytool.html.

See also

"Command line Connector" on page 37,
"z/OS TSO/E Command Line Function Component" on page 437.

z/OS TSO/E Command Line Function Component

This Function Component addresses the need Tivoli Directory Integrator to be able to issue privileged z/OS commands, including RACF, ACF2 and TopSecret commands.

Configuration

Parameters

The z/OS environment requires a number of parameters for this FC to function properly.

Partner TP Name
Specifies the Partner TP Name as specified in the APPC TP Profile. This parameter is required.

Destination LU Name
Specifies the destination LU name as specified in the APPC configuration file. If NULL or empty the LU that is defined as default will be used.

Source LU Name
Specifies the source LU name as specified in the APPC configuration file. If NULL or empty the LU that is defined as default will be used.

APPC mode
Specifies the mode of the APPC conversation. If NULL or empty the default mode as specified in the source LU will be used.

User Name
The user under whose identity the conversation will be held.

If NULL or empty, Security_Type of the conversation is **ATB_SECURITY_SAME** and the identity under which the IBM Tivoli Directory Integrator is started is used with a default profile. Otherwise, Security_Type of the conversation is **ATB_SECURITY_PROGRAM** and the TSO command will be executed under the identity of the user specified using the profile of that user.

User Password
The password of the user under whose identity the conversation will be held; only taken into account when the **User Name**parameter is specified.

If NULL or empty, the conversation will succeed only if the user specified is granted surrogate authorization on the system where the REXX script is deployed.

Detailed Log
When checked, generates additional log messages.

Comment
Your own comments go here.

Using the FC

The z/OS TSO Command Line Function Component is able to execute TSO/E shell commands.

This component is only responsible for execution of the command it is passed – it will not construct shell commands and will not understand the business logic associated with the commands it is executing.

The Function Component is given the command line on input and returns the execution status and the output generated by the command. Architecturally this FC consists of a Java layer, a USS shared library and a REXX script component: The Java layer passes the command to the shared library, the shared library passes it to the REXX script through APPC and the REXX script executes the TSO/E command and passes back the result.

Specific business logic of a higher level can be built on top of this Function Component - for example a Connector or Adapter that manages RACF users. This Connector could construct the correct RACF commands (that correspond to add, modify, and so forth) and use the FC internally to execute them.

Error Flows

The z/OS TSO Command Line FC throws an exception:

- If an error occurs while retrieving z/OS parameters.
- If it could not allocate the APPC conversation.
- If it could not execute the TSO command:

  When the following message is logged: "CTGDKB012E Could not execute TSO command. The command returned return code: 26". These could be the possible reasons for this:

  1. Service Reason: 43

     ATB80043I Calling program did not specify both a userid and a password and/or surrogate authorization check failed.

     Make sure you have specified both a valid username and password.

  2. Service Reason: 49

     ATB80049I Value specified on Local_LU_name parameter is not the name of the system base LU or the name of a NOSCHED LU

     You may look for another LU defined as "BASE=YES" in the output from the "D APPC,LU,ALL" command.

  3. Service Reason: 100

     Service: ATBRCVW

     ATB80100I From VTAM macro APPCCMD: Primary error return code: 0018, secondary error return code: 0000, sense code: 08640001.

     The specified userid is not authorized to execute the command or to execute the data set containing the TDIEXEC REXX script.

- If unable to retrieve the command output of the TSO command and TSO return code is null.
- If an error occurs while deallocating the conversation during termination.
- If any function is called with invalid parameters.

Function Component Input

An *Entry* object with an Attribute named *command* whose value is the TSO/E command to be executed.

Function Component Output

An Entry object with the following Attributes:

commandOutput
: Contains the output of the TSO/E command execution.

tsoCommandReturnCode
: Contains the return code of the TSO/E command.

appcReturnCode
: Contains the APPC return code.

Authentication

The APPC conversation can be performed in two modes: **Security_Same** and **Security_Progam**.

Whether the conversation will be held in the Security_Program mode depends on whether the **User Name** Function Component parameter contains a non NULL value.

Authorization

The REXX script is the component that actually executes the TSO command.

Tivoli Directory Integrator will be allowed or disallowed to execute the TSO command depending on the privileges of the user id specified in the z/OS TSO Command Line Function Component configuration.

To minimize the chances that the REXX script ability to execute TSO commands is maliciously exploited, the following optional deployment strategy can be applied:

A specific dataset is created for the REXX script – this dataset will contain the REXX script only and no other members. In RACF the access to the dataset will be limited to only those users that we want to allow to execute that script. The same user(s) will then need to be specified in the z/OS TSO Command Line Function Component configuration.

Other options for restricting the access to the REXX script include limiting the access provided by APPC:

- The Logical Units from which conversation requests will be accepted can be restricted. If for example the REXX script is accessed from the local system, the TP Profile can be put in a LU that is inaccessible for remote calls.
- A limited number of users might be allowed to request a conversation with the TP associated with the REXX script. For example, special users might be created that can access the TP.

Required pseudonym file

The APPC/MVS calls use pseudonyms for actual calls, characteristics, variables, and so on. For example, the return_code parameter for APPC/MVS calls can be the pseudonym atb_ok. The integer value for the atb_ok pseudonym is 0. APPC/MVS provides several pseudonym files in the header file data set `SYS1.SIEAHDR.H` that define the pseudonyms and corresponding integer values for different languages and communication calls.

The **ATBPBREX** pseudonym file is provided for APPC/MVS calls. This pseudonym file contains REXX assignment statements that simplify writing transaction programs in REXX. The TDIEXEC REXX script uses internally this pseudonym file therefore the ATBPBREX file should be available in the header file data set SYS1.SIEAHDR.H on the underlying z/OS environment.

If this file does not exist on your z/OS environment it could be copied from somewhere else or it can be created by using this sample ATBPBREX file:

```
/****START OF SPECIFICATIONS*********************************************/
/*                                                                       */
/*01* MODULE-NAME = ATBPBREX                                             */
/*                                                                       */
/*02* DESCRIPTIVE-NAME = Interface Declaration File for LU 6.2           */
/*                       Protocal Boundary Interface - REXX              */
/*                                                                       */
/*02* COMPONENT = APPC Component (SCACB)                                 */
/*                                                                       */
/*01* PROPRIETARY STATEMENT=                                             */
/****PROPRIETARY_STATEMENT************************************************/
/*                                                                       */
/*                                                                       */
/*  LICENSED MATERIALS - PROPERTY OF IBM                                 */
/*  THIS EXEC IS "RESTRICTED MATERIALS OF IBM"                           */
/*  5647-A01 (C) COPYRIGHT IBM CORP. 1998                                */
/*                                                                       */
/*  STATUS= HBB6606                                                      */
/*                                                                       */
/*  EXTERNAL CLASSIFICATION:  GUPI                                       */
/*                                                                       */
/*  END OF EXTERNAL CLASSIFICATION                                       */
/*                                                                       */
/****END_OF_PROPRIETARY_STATEMENT**************************************/
/*                                                                       */
/*                                                                       */
/*01* DISCLAIMER =                                                       */
```

```
/*                                                                   */
/*   THIS SAMPLE SOURCE IS PROVIDED FOR TUTORIAL PURPOSES ONLY. A    */
/*   COMPLETE HANDLING OF ERROR CONDITIONS HAS NOT BEEN SHOWN OR     */
/*   ATTEMPTED, AND THIS SOURCE HAS NOT BEEN SUBMITTED TO FORMAL IBM */
/*   TESTING. THIS SOURCE IS DISTRIBUTED ON AN 'AS IS' BASIS         */
/*   WITHOUT ANY WARRANTIES EITHER EXPRESSED OR IMPLIED.             */
/*                                                                   */
/*01* FUNCTION = LU 6.2 REXX pseudonym file                         */
/*                                                                   */
/*01* METHOD OF ACCESS:                                             */
/*                                                                   */
/*  If you are using interpreted REXX provided by TSO/E the          */
/*  EXECIO command should be used to read this file.                 */
/*                                                                   */
/*01* DISTRIBUTION LIBRARY: AIEAHDR                                 */
/*                                                                   */
/*01* CHANGE-ACTIVITY:                                              */
/*                                                                   */
/*  FLAG LINEITEM  FMID    DATE  ID  COMMENT                         */
/*  $01=OY54027  HBB4420 920505 PDI8: MAKE PART AVAILABLE IN HBB4420 */
/*  $P1=PKB0817  HBB4430 920729 PDI8: Support of Conversation State  */
/*      constants.                                                   */
/*  $L1=APPCP    HBB6603 960105 PDE6: APPC/MVS PC support            */
/****END OF SPECIFICATIONS*********************************************/
  /* ******************************************************************/
  /*  Conversation State Values                                @P1A*/
  /* ******************************************************************/
     atb_initialize_state           = 2                      /*@P1A*/
     atb_send_state                 = 3                      /*@P1A*/
     atb_receive_state              = 4                      /*@P1A*/
     atb_send_pending_state         = 5                      /*@P1A*/
     atb_confirm_state              = 6                      /*@P1A*/
     atb_confirm_send_state         = 7                      /*@P1A*/
     atb_confirm_deallocate_state   = 8                      /*@P1A*/
     atb_defer_receive_state        = 9                      /*@L1A*/
     atb_defer_deallocate_state     = 10                     /*@L1A*/
     atb_sync_point_state           = 11                     /*@L1A*/
     atb_sync_point_send_state      = 12                     /*@L1A*/
     atb_sync_point_dealloc_state   = 13                     /*@L1A*/

  /* ******************************************************************/
  /*  Conversation Type Values                                        */
  /* ******************************************************************/
     atb_basic_conversation = 0
     atb_mapped_conversation = 1
  /* ******************************************************************/
  /*  Data Received Values                                            */
  /* ******************************************************************/
     atb_no_data_received = 0
     atb_data_received = 1
     atb_complete_data_received = 2
     atb_incomplete_data_received = 3
  /* ******************************************************************/
  /*  Deallocate Type Values                                          */
  /* ******************************************************************/
     atb_deallocate_sync_level = 0
     atb_deallocate_flush =1
     atb_deallocate_confirm = 2
     atb_deallocate_abend   = 3
  /* ******************************************************************/
  /*  Error Direction Values                                          */
  /* ******************************************************************/
     atb_receive_error      = 0
     atb_send_error         = 1
  /* ******************************************************************/
  /*  Fill Values                                                     */
  /* ******************************************************************/
```

```
   atb_fill_ll            = 0
   atb_fill_buffer        = 1
/* ******************************************************************/
/*  Lock Values                                                     */
/* ******************************************************************/
   atb_locks_short        = 100
   atb_locks_long         = 101
/* ******************************************************************/
/*  Prepare to Receive Type Values                                  */
/* ******************************************************************/
   atb_prep_to_receive_sync_level  = 0
   atb_prep_to_receive_flush       = 1
   atb_prep_to_receive_confirm     = 2
/* ******************************************************************/
/*  Notify Type Values                                              */
/* ******************************************************************/
   atb_notify_type_none   = '00000000'X
   atb_notify_type_ecb    = '00000001'X
/* ******************************************************************/
/*  Request To Send Received Values                                 */
/* ******************************************************************/
   atb_req_to_send_not_received    = 0
   atb_req_to_send_received        = 1
/* ******************************************************************/
/*  Return Code Values                                              */
/* ******************************************************************/
   atb_ok                 = 0
   atb_allocate_failure_no_retry   = 1
   atb_allocate_failure_retry      = 2
   atb_conversation_type_mismatch  = 3
   atb_pip_not_specified_correctly = 5
   atb_security_not_valid = 6
   atb_sync_lvl_not_supported_lu   = 7                      /*@L0A*/
   atb_sync_lvl_not_supported_pgm  = 8
   atb_tpn_not_recognized = 9
   atb_tp_not_available_no_retry   = 10
   atb_tp_not_available_retry      = 11
   atb_deallocated_abend           = 17
   atb_deallocated_normal = 18
   atb_parameter_error    = 19
   atb_product_specific_error      = 20
   atb_program_error_no_trunc      = 21
   atb_program_error_purging       = 22
   atb_program_error_trunc         = 23
   atb_program_parameter_check     = 24
   atb_program_state_check         = 25
   atb_resource_failure_no_retry   = 26
   atb_resource_failure_retry      = 27
   atb_unsuccessful                = 28
   atb_deallocated_abend_svc       = 30
   atb_deallocated_abend_timer     = 31
   atb_svc_error_no_trunc          = 32
   atb_svc_error_purging           = 33
   atb_svc_error_trunc             = 34
   atb_take_backout                 = 100                   /*@L0A*/
   atb_deallocated_abend_bo         = 130                   /*@L0A*/
   atb_deallocated_abend_svc_bo     = 131                   /*@L0A*/
   atb_deallocated_abend_timer_bo   = 132                   /*@L0A*/
   atb_resource_fail_no_retry_bo    = 133                   /*@L0A*/
   atb_resource_failure_retry_bo    = 134                   /*@L0A*/
   atb_deallocated_normal_bo        = 135                   /*@L0A*/
/* ******************************************************************/
/*  Reason Code Values                                        @L0A*/
/* ******************************************************************/
   atb_invalid_vote_read_only      = 1                      /*@L0A*/
   atb_invalid_wait_for_outcome    = 2                      /*@L0A*/
   atb_invalid_action_if_problems  = 3                      /*@L0A*/
```

```
   atb_extract_exit_not_specified  = 4                       /*@L0A*/
   atb_extract_exit_failed         = 5                       /*@L0A*/
   atb_no_active_tp                = 6                       /*@L0A*/
   atb_service_error               = 7                       /*@L0A*/
/* ******************************************************************/
/*  Return Control Values                                           */
/* ******************************************************************/
   atb_when_session_allocated      = 0
   atb_immediate          = 1
   atb_when_conwinner_allocated    = 100
/* ******************************************************************/
/*  Security Type Values                                            */
/* ******************************************************************/
   atb_security_none      = 100
   atb_security_same      = 101
   atb_security_program   = 102
/* ******************************************************************/
/*  Send Type Values                                                */
/* ******************************************************************/
   atb_buffer_data        = 0
   atb_send_and_flush     = 1
   atb_send_and_confirm   = 2
   atb_send_and_prep_to_receive    = 3
   atb_send_and_deallocate         = 4
/* ******************************************************************/
/*  Status Received Values                                          */
/* ******************************************************************/
   atb_no_status_received = 0
   atb_send_received      = 1
   atb_confirm_received   = 2
   atb_confirm_send_received       = 3
   atb_confirm_dealloc_received    = 4
   atb_take_syncpt                 = 5                     /* @L0A*/
   atb_take_syncpt_send            = 6                     /* @L0A*/
   atb_take_syncpt_dealloc         = 7                     /* @L0A*/
/* ******************************************************************/
/*  Sync Level Values                                               */
/* ******************************************************************/
   atb_none               = 0
   atb_confirm            = 1
   atb_syncpt             = 2                             /* @L0A*/
/* ******************************************************************/
/*  Set Syncpt Options Values                                 @L0A*/
/* ******************************************************************/
   atb_syncpt_options_nochange     = 0                      /*@L0A*/
   atb_vote_read_only_no           = 1                      /*@L0A*/
   atb_vote_read_only_yes          = 2                      /*@L0A*/
   atb_wait_for_outcome_no         = 1                      /*@L0A*/
   atb_wait_for_outcome_yes        = 2                      /*@L0A*/
   atb_action_if_problems_commit   = 1                      /*@L0A*/
   atb_action_if_problems_backout  = 2                      /*@L0A*/
```

If you decide to create the file, a new FB 80 z/OS data set has to be allocated, and the TDIEXEC script has to be edited to use a different data set. If you have created a data set named, for example, ROOT.MYATBREX, the TDIEXEC should be edited like this:

```
...
/*********************************************************************/
 /*  Get psuedonym definition file for REXX for the LU6.2 Verbs      */
 /*  sys1.sieahdr.h(atbpbrex)                                        */
 /*********************************************************************/

"alloc f(datain) da('ROOT.MYATBREX') shr reuse"
"execio * diskr datain (stem  linelist.  finis"
do x = 1 to linelist.0
  interpret linelist.x
```

```
end
drop linelist.
"free f(datain)"
...
```

Note: The provided ATBPBREX file is just an example and may not be applicable on every z/OS environment.

Setting up the native part of the FC

Before using the TSO Command Line Function Component, a REXX script must be deployed on a z/OS dataset and APPC configured accordingly:

The z/OS TSO Command Line Function Component contains a REXX script named TDIEXEC that executes a TSO/E command and returns the command output.

This REXX script has to be copied to a FB 80 z/OS dataset where it will be invoked from.

The z/OS TSO Command Line Function Component contains a JCL named TDITP.jcl that defines the TP Profile data for the REXX script.

You customize the JCL according to the z/OS system environment and execute it.

In detail, in order to deploy the FC you should:

1. Identify (or allocate) PDS datasets where the JCL and REXX script will reside. The JCL and the REXX script can reside in the same or in different datasets.

   You can find the TDITP.jcl JCL and TDIEXEC REXX script in the tso_fc subfolder of the IBM Tivoli Directory Integrator installation folder (only on z/OS).
2. 2. Copy the REXX script and the JCL from the sample library to the already created PDS dataset.

   For example, this can be done from the TSO shell or menu 6 of ISPF with the following commands:

   ```
   OGET 'TDI_install_dir/tso_fc/TDIEXEC' '<TDIEXEC_dataset>(TDIEXEC)'
   OGET 'TDI_install_dir/tso_fc/TDITP.jcl' '<TDITP.jcl_dataset>(TDITP)'
   ```
3. Customize TDITP.jcl to reflect your environment.

   To do so, follow the instructions within the TDITP.jcl JCL. Basically you need to specify these names:

   - Name of the data set where the TDIEXEC REXX script resides – specify the name of the identified/allocated data set in step 1.
   - Name of the APPC TP Profile data set – each APPC Transaction Program (TP) has a TP Profile defined to APPC/MVS. These definitions are stored in the APPC TP Profile data set. The default name of this dataset is SYS1.APPCTP, but it can be customized by the installation.
   - Name of The Transaction Scheduler defined class – By default the ASCH configuration file is located in the USER.PARMLIB dataset. Browse the PARMLIB member named ASCHPMxx (where 'xx' can vary, for example, ASCHPM00 or ASCHPM1A) and use the name of any class that is already defined.

     You may also create your own class of transaction initiators by adding a similar definition – 'CLASSADD CLASSNAME(MYCLASS) MSGLIMIT(1000) MAX(10) MIN(1) RESPGOAL(1)'. The definition needs to be activated with the 'SET ASCH=xx' system command in which the 'xx' are the last two characters of the ASCHPMxx USER.PARMLIB member.

     Note: The minimum number of started transaction initiators should correspond to the expected number of transactions running at a time.
4. Make sure APPC and ASCH (Transaction Scheduler) are started – verify that the APPC and ASCH address spaces for APPC/MVS and the transaction scheduler are active on the system. This can be checked by executing the following commands: 'D APPC' and 'D ASCH'.

If either or both of them are not running, they need to be started using these commands: 'S APPC,SUB=MSTR,APPC=xx' and 'S ASCH,SUB=MSTR,ASCH=xx'. The 'xx' are the last two characters of the APPCPMxx and ASCHPMxx USER.PARMLIB members.

5. Make sure the specified LUs exist and are active - the destLuName and srcLuName parameters must specify existing LUs or if set to null a default LU must be defined.

   This can be ensured by checking the APPC configuration file which by default is located in the USER.PARMLIB data set. The PARMLIB member named APPCPMxx (where 'xx' can vary, for example, APPCCPM00 or APPCPM1A) contains definitions of all local LUs. The base logical unit for transaction scheduler is marked with 'BASE'.

   Note: The LU names defined in that file must correspond to the VTAM application definitions for APPC/MVS located in the USER.VTAMLST members.

   For example here are the definitions of the BASELU logical unit:

   File: USER.PARMLIB(APPCPM00)

```
LUADD
  ACBNAME(BASELU)
  BASE
  SCHED(ASCH)
  TPDATA(SYS1.APPCTP)
  TPLEVEL(SYSTEM)
SIDEINFO
  DATASET(SYS1.APPCSI)
```

   File: USER.VTAMLST(A01APPC)

```
         VBUILD TYPE=APPL
BASELU   APPL  ACBNAME=BASELU,                                  X
               APPC=YES,                                        X
               AUTOSES=0,                                       X
               DDRAINL=NALLOW,                                  X
               DLOGMOD=#BATCH,                                  X
               DMINWNL=5,                                       X
               DMINWNR=5,                                       X
               DRESPL=NALLOW,                                   X
               DSESLIM=10,                                      X
               LMDENT=19,                                       X
               MODETAB=LOGMODES,                                X
               PARSESS=YES,                                     X
```

   Note: Even if a LU is defined it still may not be active. You can check if a LU is actually active by executing the following system command: 'D APPC,LU,ALL', which will display information about all LUs. To activate a particular LU use the VTAM command VARY ACT (for example, 'V NET,ACT,ID=A01APPC', where 'A01APPC' is the name of the VTAMLST member).

6. Submit the modified TDITP.jcl.

   You can submit it from ISPF by typing 'sub' in front of the name of the JCL.

Notes:

1. If you want to execute the system commands from ISPF, go to "System Display and Search Facility", Menu 's' and prefix your command with '/' (for example, '/s APPC,SUB=MSTR').
2. If any of the mentioned data sets does not exist on your system please contact your systems programmer or your network administrator for assistance.

See also

"z/OS environment Support", in *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*.

Chapter 5. SAP ABAP Application Server Component Suite

Who should read this chapter

IBM Tivoli Directory Integrator components are designed for network administrators who are responsible for maintaining user directories and other resources. This chapter assumes that you have practical experience installing and using both IBM Tivoli Directory Integrator and SAP ABAP Application Server, and it describes the procedural steps that are required to achieve integration between IBM Tivoli Directory Integrator and SAP ABAP Application Server.

This chapter assumes that both IBM Tivoli Directory Integratorand SAP ABAP Application Server are installed, configured and running on your network. No details are provided regarding the installation and configuration of these products, except where necessary to achieve integration.

Component Suite Installation

This section describes the software requirements and installation steps for the IBM Tivoli Directory Integrator Component Suite for SAP ABAP Application Server.

This chapter contains the following sub-sections:

- "Software Requirements"
- "Verifying the Component Suite for SAP ABAP Application Server" on page 446
- "Checking the Version Numbers" on page 447
- "Uninstallation" on page 448

Software Requirements

Installing IBM Tivoli Directory Integrator 7.1 also installs the Component Suite. However, to complete the install of the Component Suite, an additional component must be added on the target machine:

- SAP Java Connector (JCo) version 2.1.6 or above

The IBM Tivoli Directory Integrator Component Suite for SAP ABAP Application Server is supported on the operating systems platforms that are common for IBM Tivoli Directory Integrator and SAP JCo. Please see the IBM Tivoli Directory Integrator Administrators Guide for supported operating system platforms supported by IBM Tivoli Directory Integrator and please see the SAP website for information about supported platforms for SAP JCo. SAP JCo has other prerequisites, including the following:

Windows
: The SAP JCo libraries require the MS 8.0 C/C++ runtime. See SAP Note 684106 for instructions. As a workaround msvcr71.dll, msvcp71.dll, mfc71.dll, mfc71u.dll etc may be copied from other Windows computers (32 and 64-bit versions of the DLLs are available, the version you need to copy must match the version of the SAP DLLs you have.)

Linux
: The latest versions of libstdc++, libgcc, and compat-libstdc++ may be required. Information related to C++ Runtime 6.0 (libstdc++.so.6) can be found in SAP Note 1021236.

Licensed SAP ABAP Application Server customers can download the JCo from the SAP Website. You will require a valid SAP support login account and password, which can be obtained by request from SAP support. A supported version of SAP ABAP Application Server must also be installed and running on a node within the network environment. TCP/IP network connectivity is required between the SAP ABAP Application Server instance and the machine hosting the IBM Tivoli Directory Integrator Component Suite for SAP ABAP Application Server.

Supported versions of SAP ABAP Application Server are:

- SAP ABAP Application Server v6.20
- SAP ABAP Application Server v6.40
- SAP ABAP Application Server v7.0

Configuring the SAP Java Connector

Once downloaded and available on the machine designated to host IBM Tivoli Directory Integrator and the Component Suite for SAP ABAP Application Server, the JCo can be installed and configured for IBM Tivoli Directory Integrator 7.1 as follows:

1. Unzip the JCo distribution package to a directory on the target machine. For example:
   `/SapJco216`
2. Open the `installation.html` file and follow the installation instructions for your Operating System. For example:
   `/SapJco216/docs/jco/installation.html`
3. Add the following entries to your network service file:
   - `sapdpNN 32NN/tcp`
   - `sapgwNN 33NN/tcp`

   - where *NN* is the SAP instance identifier of the SAP system to which the IBM Tivoli Directory Integrator Component Suite for SAP ABAP Application Server will connect.
4. Copy `sapjco.jar` from the SAP JCo package directory to the *Tivoli Directory Integrator_HOME*`/jars` folder.
5. If you intend to use the "ALE Intermediate Document (IDOC) Connector for SAP ABAP Application Server and SAP ERP" on page 471, you need to copy `sapidoc.jar` and `sapidocjco.jar` to the same location as well.
6. **On Windows machines only,** copy `librfc32.dll` and `sapjcorfc.dll` to the *Tivoli Directory Integrator_HOME*`/libs` folder.

Notes:

1. The network service file can be found at `%system_root%\system32\drivers\etc\services` on Windows 32, or `/etc/services` on UNIX.
2. Before using the IBM Tivoli Directory Integrator Component Suite for SAP ABAP Application Server, ensure that the `sapjco.jar` is in the CLASSPATH, and that `sapjcorfc.{dll/so}` and `librfc*.{dll/so}` are in the loadable library path.

Verifying the Component Suite for SAP ABAP Application Server

To verify the IBM Tivoli Directory Integrator 7.1 Component Suite for SAP ABAP Application Server:

Table 67 below describes the files and locations installed by the system installer for IBM Tivoli Directory Integrator 7.1, with regards to the Components Suite.

Table 67. Installed locations for the IBM Tivoli Directory Integrator Component Suite

Filename	Description
`SapR3BorConnector.jar`	*Tivoli Directory Integrator_HOME*`/jars/connectors`
`SapR3UserConnector.jar`	*Tivoli Directory Integrator_HOME*`/jars/connectors`
`SapR3RfcFC.jar`	*Tivoli Directory Integrator_HOME*`/jars/functions`
`index.html` (Javadoc for all SAP Components)	*Tivoli Directory Integrator_HOME*`/docs/api/`
`bapi_user_actgroups_assign.xsl`	*Tivoli Directory Integrator_HOME*`/xsl`
`bapi_user_actgroups_delete.xsl`	*Tivoli Directory Integrator_HOME*`/xsl`
`bapi_user_change.xsl`	*Tivoli Directory Integrator_HOME*`/xsl`

Table 67. Installed locations for the IBM Tivoli Directory Integrator Component Suite (continued)

Filename	Description
bapi_user_create.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_user_delete.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_user_getdetail_postcall.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_user_getdetail_precall.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_user_getlist_postcall.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_user_getlist_precall.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_user_profiles_assign.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_user_profiles_delete.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_employee_dequeue.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_employee_enqueue.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_employee_getdata_postcall.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_employee_getdata_precall.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_persdata_change.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_persdata_create.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_persdata_delete.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_persdata_getdetail_postcall.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_persdata_getdetail_precall.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_persdata_getdetailedlist_postcall.xsl	*Tivoli Directory Integrator_HOME*/xsl
bapi_persdata_getdetailedlist_precall.xsl	*Tivoli Directory Integrator_HOME*/xsl

Note: The Connectors rely on the XSL stylesheets to perform their operations and by default the Connectors will locate the stylesheets by using a relative path, for example, xsl/bapi_user_getlist_precall.xsl . It is important to be aware of this default reliance on using a relative path if you are using a Tivoli Directory Integrator Solution directory. As a result, you will need to do one of the following:

1. Copy the *TDI_install_dir*/xsl folder into your Tivoli Directory Integrator Solution directory.
2. Set your Solution directory to be the Tivoli Directory Integrator install directory.
3. Elect not to configure a Solution directory.

If the XSL folder is not available in the Tivoli Directory Integrator Solution directory then an error similar to the following will result when attempting to use the SAP Connectors:

```
com.ibm.di.connector.sapr3.user.UserRegistryConnectorException: CTGDIK019E The Connector detected an
 exception during initialization. The message is: 'CTGDIK008E The configured XSL file named
 'xsl/bapi_user_getdetail_precall.xsl' does not exist.'.
```

Checking the Version Numbers

To check the component software version numbers for this integration package:

1. Start IBM Tivoli Directory Integrator and click on **Help**
2. Select **About IBM Tivoli Directory Integrator Components**.
3. Version numbers are displayed for the following components:
   - **ibmdi.SapR3RfcFC**
   - **ibmdi.SapR3UserRegConnector**
   - **ibmdi.SapR3BorConnector**

Uninstallation

To remove the IBM Tivoli Directory Integrator the Component Suite for SAP ABAP Application Server from the target system:

1. Stop IBM Tivoli Directory Integrator assembly lines that are currently running and using one of the IBM Tivoli Directory Integrator Components for SAP ABAP Application Server.
2. Run the uninstall executable located at *Tivoli Directory Integrator_HOME*/_uninstsap and follow the prompts.
3. Remove the following entries from your network service file (%system_root%\system32\drivers\etc\services on Windows 32, /etc/services on UNIX):
   - sapdp*NN* 32*NN*/tcp
   - sapgw*NN* 33*NN*/tcp

   - where *NN* is the SAP instance identifier of the SAP system to which the IBM Tivoli Directory Integrator Component Suite for SAP ABAP Application Server connects.
4. Remove the SAP JCo (*SAP_JCO_HOME*) directory that was created during the installation.
5. Remove the environment variable entries and additions that were created during installation as a result of following the instructions within *SAP_JCO_HOME*/docs/jco/installation.html.
6. Remove sapjco.jar from the *TDI_install_dir*/jars folder.
7. **On Windows machines only,** remove the librfc32.dll and sapjcorfc.dll files from the *Tivoli Directory Integrator_HOME*/libs folder.

Function Component For SAP ABAP Application Server

This chapter describes the IBM Tivoli Directory Integrator Function Component for SAP ABAP Application Server.

This chapter includes the following sections:

- "Function Component Introduction"
- "Configuration"
- "Using the Function Component" on page 451

Function Component Introduction

The Function Component for SAP ABAP Application Server uses SAP JCo to invoke RFCs on the SAP ABAP Application Server System. The Function Component provides a means of calling an arbitrary RFC.

Figure 1 below illustrates the overview architecture of the RFC Function Component.

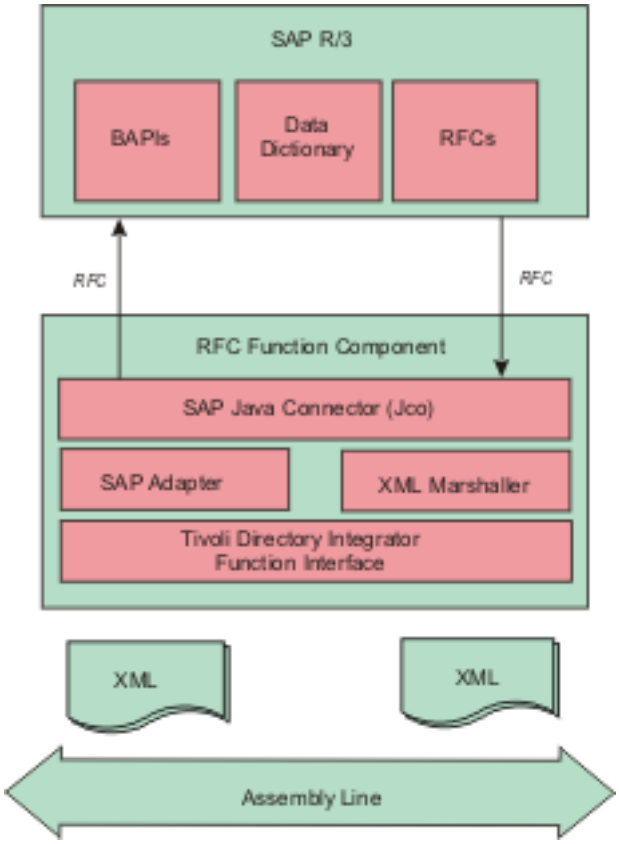


Figure 1. Overview architecture of the RFC Function Component

Before using the Function Component for SAP ABAP Application Server, the SAP JCo must be downloaded and installed (for details, see "Software Requirements" on page 445).

Configuration

If the Function Component for SAP ABAP Application Server (SAP ABAP AS) is added directly into an assembly line, the following configuration parameters are available for client connections. The parameters are very similar to the logon parameters for the traditional SAP GUI. Runtime names are shown below in parentheses.

Parameters

ABAP AS Client (client)
SAP ABAP AS Logon client for SAP connection. For example, 100.

ABAP AS User (user)
SAP ABAP AS Logon user for SAP connection.

Password (passwd)
SAP ABAP AS Logon password for SAP connection.

ABAP AS System Number (sysnr)
The SAP ABAP AS system number for SAP connection. For example, 00.

ABAP AS Hostname (ashost)
SAP ABAP AS application server name for SAP connection.

Gateway host (gwhost)
Gateway host name for SAP connection.

RFC Trace (trace)
Set to one (*1*) to enable RFC API tracing. If enabled, the SAP RFC API will produce separate `rfc_nnnn.trc` files (where *nnnn* represents values assigned by the RFC API) in the working directory IBM Tivoli Directory Integrator This option may be useful to help diagnose RFC invocation problems. It logs the activity and data between the Connector and SAP ABAP AS. This should be set to zero (*0*) for production deployment.

Additional configuration parameters are available when using the Function Component programmatically. For more information on the additional parameters, see the `SapR3RfcFC` Java Doc in the distribution package.

Function Component Input

The **perform()** method accepts an **Entry** object. If anything else is passed an Exception is thrown. The **Entry** object contains two attributes:

- **requestType**
- **request**

The Function Component supports three styles of invocation:

- XML Document,
- XML string, or
- multi-valued attribute.

The value of **requestType** should be set to one of the following, to indicate which style is to be used:

- *xmlDomDocument*
- *xmlString*
- *multiValuedAttributes*

The value of attribute **request** is a type of:

- `org.w3c.dom.Document` if **requesttype** is *xmlDomDocument*,
- `java.lang.String` if **requesttype** is *xmlString*, or
- `com.ibm.di.entry.Attribute` if **requesttype** is *multiValuedAttributes*.

The value of **request** represents the request data of an RFC as one of:

- XML String,
- DOM Document, or
- multi-valued Attribute (please refer to the Javadoc for some sample JavaScript using multi-values attribute invocation).

Any other value will result in an Exception being thrown.

If request is of type `org.w3c.dom.Document`:
Its associated value must be an `org.w3c.dom.Document` containing an XSchema that conforms to the specification for ABAP RFC XML serialization.

If request is of type `java.lang.String`:
Its associated value must be an XML string. A DOM parser will parse the string value. Its XSchema must also conform to the specification for Serialization of ABAP Data in XML.

If request is a multi-valued attribute:
The first value of attribute **request** must be of type `java.lang.String`, containing the name of the RFC, while the second value of the attribute **request** must be `com.ibm.di.entry.Attribute`, whose values contain additional attributes for the SAP RFC parameters as a series of nested and multi-valued attributes representing the names of the import and table parameters of the RFC. The names of the parameters must be encoded according to the rules for Serialization of ABAP Data in XML (names will not have characters that could result in badly-formed XML).

Here is an example of how to invoke the Function Component using the multi-valued attributes style:

```
var rfc = system.newAttribute("BAPI_SALESORDER_GETLIST");
var attr1 = system.newAttribute("CUSTOMER_NUMBER");
attr1.addValue("0000000016");
var attr2 = system.newAttribute("SALES_ORGANIZATION");
attr2.addValue("AU01");
rfc.addValue(attr1);
rfc.addValue(attr2);
var entry = system.newEntry();
var reqAttr = entry.newAttribute("request");
reqAttr.addValue(rfc);
entry.setAttribute("requestType", "multiValuedAttributes");
var result = fc.perform(entry);
```

Function Component Output

The Function Component output is an **Entry** object with two attributes:

- **responseType**, indicating the response type,
- **response**, with the RFC response as either a DOM Document, an XML string or a nested multi-valued Attribute.

Attribute **responseType** will have a `java.lang.String` value corresponding to the input request type.

If the Entry contains an attribute responseType with value *xmlDomDocument*
The value of attribute **response** is an `org.w3c.dom.Document` containing the RFC response.

If the Entry contains an attribute responseType with value *xmlString*
The value of attribute **response** is an XML `java.lang.String` containing the RFC response.

If the Entry contains an attribute responseType with value *multiValuedAttributes*
The value of attribute **response** is a nested and multi-valued attribute, where the first value is a `java.lang.String`, which has the name of the RFC that was invoked, and the second value contains the results of the RFC as a series of nested multi-valued attributes.

Using the Function Component

The Function Component invokes the given RFC for a SAP ABAP Application Server system.

It can be placed in an assembly line or invoked directly from script. It is the callers' responsibility to check the returned Entry object for any errors that may have resulted from invoking the RFC.

As an example, the following code can be used to invoke an RFC from JavaScript:

```
var counter = 0;
var fc = system.getFunction("ibmdi.SapR3RfcFC");
var myentry;
var docResponse;

fc.setParam(fc.PARAM_CONFIG_CLIENT, "100");
fc.setParam(fc.PARAM_CONFIG_USER, "TIVOLI");
fc.setParam(fc.PARAM_CONFIG_PASSWORD, "******");
fc.setParam(fc.PARAM_CONFIG_SYSNUMBER, "11");
fc.setParam(fc.PARAM_CONFIG_LANGUAGE, "E");
fc.setParam(fc.PARAM_CONFIG_APPLICATION_SERVER, "kimala");

fc.initialize(null);
var rfc = new java.lang.String("<BAPI_COMPANYCODE_GETLIST/>");
var myentry = system.newEntry();
var attr = myentry.newAttribute(fc.PARAM_INPUT_TYPE);
attr.addValue(fc.PARAM_VAL_STRING);

attr = myentry.newAttribute(fc.PARAM_INPUT);
attr.addValue(rfc);
var myresponse = fc.perform(myentry);

//system.dumpEntry(myresponse);
fc.terminate();
```

Note: Configuration parameters must be set before **initialize()** is called, and **terminate()** should be called to cleanup.

Using the Command Line RFC Invoker

As a tool to assist in creating valid RFC XML requests, a command line utility has been provided. It can be invoked outside of the IBM Tivoli Directory Integrator environment and is able to read an XML file, which represents an RFC XML request to be executed against the SAP ABAP Application Server system.

To invoke the utility, first add the following jars to the CLASSPATH environment variable:

- *TDI_install_dir*/jars/functions/SapR3RfcFC.jar
- *TDI_install_dir*/jars/common/tdiresource.jar
- *TDI_install_dir*/jars/3rdparty/IBM/icu4j_4_2.jar

Then, invoke using the command:

```
TDI_install_dir/jvm/bin/java com.ibm.di.fc.sapr3rfc.RfcXmlInvoker -f
   [input XML file] -o [output XML file] -p
   [JCO Connection properties file]
```

Notes:

1. These instructions assume that you have already completed the steps described in "Configuring the SAP Java Connector" on page 446. It is important that the sapjco.jar is in the CLASSPATH, and that sapjcorfc.{dll/so} and librfc*.{dll/so} are in the loadable library path.
2. For AIX, the path to the Java executable is *TDI_install_dir*/jvm/jre/bin/java.exe

The contents of the JCO Properties file represent the SAP ABAP AS client connection parameters for the SAP system. An example of the values in the property file is shown below:

```
jco.client.client=R/3 CLIENT
jco.client.user=R/3 USER NAME
jco.client.passwd=R/3 USER PASSWORD
jco.client.sysnr=R/3 SYSTEM NUMBER
jco.client.ashost=R/3 APPLICATION SERVER HOSTNAME OR IP ADDRESS
jco.client.trace=RFC API TRACE: 1 = ON; 0 = OFF
```

User Registry Connector for SAP ABAP Application Server

The section describes the configuration and operation of the IBM Tivoli Directory Integrator User Registry Connector for SAP ABAP Application Server.

This chapter contains the following sections:

- "Introduction"
- "Configuration" on page 454
- "Using the User Registry Connector for SAP ABAP Application Server" on page 457

This component is not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.

Introduction

This component enables the provisioning and management of SAP user accounts to external applications (with respect to SAP ABAP Application Server). The Connector uses the generic RFC invocation feature of the IBM Tivoli Directory Integrator Function Component for SAP ABAP Application Server (referred to hereafter as the RFC Function Component). The RFC Function Component enables the Connector to manage SAP user account attributes by executing RFC ABAP code as an external SAP ABAP Application Server client application.

The Connector supports an extendable generic framework for provisioning SAP user accounts and their associated attributes. This is achieved by defining an XML representation of user account information. This XML is then transformed via XSL style sheet transformations (XSLT) into RFC requests. The default functionality of the Connector does not require the deployment of custom RFC ABAP code onto the target SAP ABAP Application Server instance.

The key features and benefits of the Connector are:

- Support for Create, Read, Update, and Delete (C.R.U.D) operations for SAP users.
- Modifiable behavior through XSL transformations for SAP ABAP Application Server RFC execution.
- Minimal compile time dependency between the Connector and SAP ABAP Application Server. The Connector does not use any generated RFC proxy code. It relies on the RFC Function Component as a dynamic proxy.

The Connector supports the following IBM Tivoli Directory Integrator Connector modes:

- Add Only
- Update
- Delete
- Lookup
- Iterator

Figure 2 below illustrates the component design of the SAP User Registry.

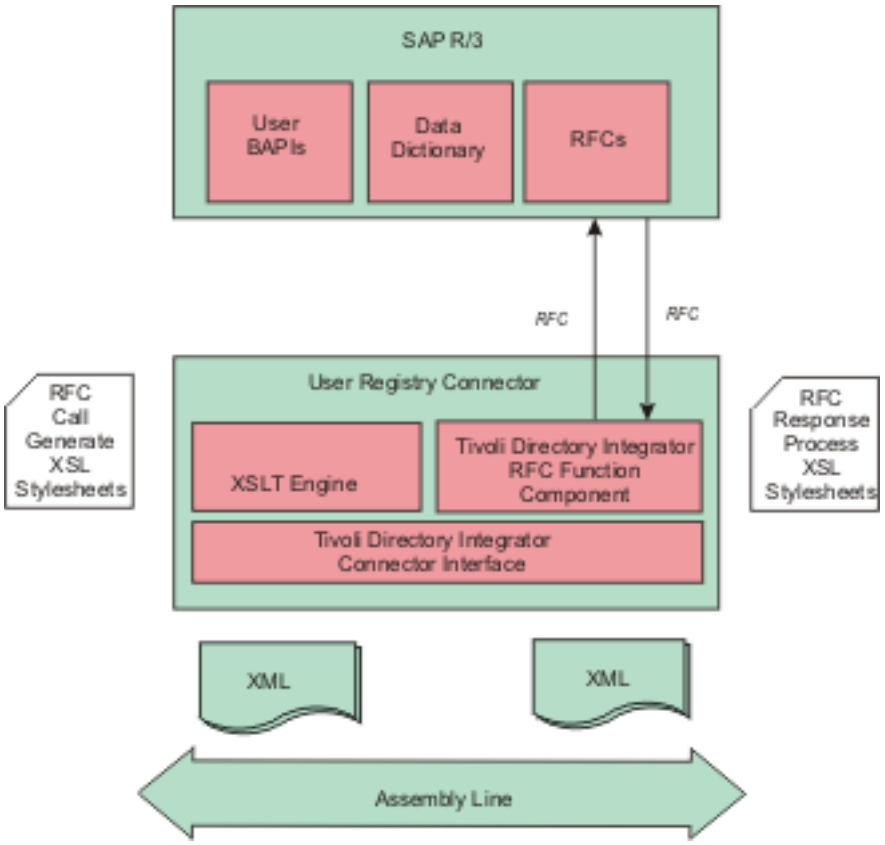


Figure 2. Component design of the SAP User Registry

Skip Lookup in Update and Delete mode

The User Registry Connector for SAP ABAP Application Server supports the **Skip Lookup** general option in Update or Delete mode. When it is selected, no search is performed prior to actual update and delete operations.

For this to function, the **sapUserName** attribute should be defined in the Link Criteria of the Connector.

Configuration

The User Registry Connector for SAP ABAP Application Server (SAP ABAP AS) may be added directly into an assembly line. The following section lists the configuration parameters that are available for SAP ABAP Application Server client connections and XSL style sheet behavior. The runtime names are shown in parentheses.

Parameters

ABAP AS Client (client)
SAP ABAP AS Logon client for SAP connection (for example, *100*). This is passed directly to the RFC Function Component.

ABAP AS User (user)
SAP ABAP AS Logon user for SAP connection. This is passed directly to the RFC Function Component.

Password (passwd)
SAP ABAP AS Logon password for SAP connection. This is passed directly to the RFC Function Component.

ABAP AS System Number (sysnr)
The SAP ABAP AS system number for SAP connection (for example, *100*). This is passed directly to the RFC Function Component.

ABAP AS Hostname (ashost)
SAP ABAP Application Server name for SAP connection. This is passed directly to the RFC Function Component.

Gateway host (gwhost)
Gateway host name for SAP connection. This is passed directly to the RFC Function Component.

RFC Trace (trace)
Set to one (*1*) to enable RFC API tracing. If enabled, the SAP RFC API will produce separate `rfc_nnnn.trc` files in the working directory of IBM Tivoli Directory Integrator. This option may be useful to help diagnose RFC invocation problems. It logs the activity and data between the Connector and SAP ABAP Application Server. This should be set to zero (*0*) for production deployment.

Optional RFC Connection Parameters
Used to define a list of other optional RFC connection parameters. The value for this configuration list is a key=value list where each connection parameter is separated by the space character. For example the following string value would set the SAP Gateway Service to "sapgw00" and enable the SAP GUI.

```
"gwserv=sapgw00 use_sapgui=1"
```

Here is a list of optional SAP Java Connector parameters that are accessible.

Alias user name (alias_user)
SAP message server (mshost)
Gateway service (gwserv)
Logon language (lang)
1 (Enable) or 0 (disable) RFC trace (trace)
Initial codepage in SAP notation (codepage)
Secure network connection (SNC) mode, 0 (off) or 1 (on) (snc_mode)
SNC partner, for example, p:CN=R3, O=XYZ-INC, C=EN (snc_partnername)
SNC level of security, 1 to 9 (snc_qop).
SNC name. Overrides default SNC partner (snc_myname)
Path to library which provides SNC service (snc_lib)
SAP R/3 name (r3name)
Group of SAP application servers (group)
Program ID of external server program (tpname)
Host of external server program (tphost)
Type of remote host 2 = R/2, 3 = R/3, E = External (type)
Enable ABAP debugging 0 or 1 (abap_debug)
Use remote SAP graphical user interface (0/1/2) (use_sapgui)
Get/Don't get a SSO ticket after logon (1 or 0) (getsso2)
Use the specified SAP Cookie Version 2 as logon ticket (mysapsso2)
Use the specified X509 certificate as logon ticket (x509cert)
Enable/Disable logon check at open time, 1 (enable) or 0 (disable) (lcheck)
String defined for SAPLOGON on 32-bit Windows (saplogon_id)
Data for external authentication (PAS) (extiddata)
Type of external authentication (PAS) (extidtype)

Idle timeout (in seconds) for the connection after which it will be closed by R/3. Only positive values are allowed. (idle_timeout)
Enable (1) or Disable (0) dsr support (dsr)

RFC Function Component Name (sapr3.userconn.rfcFC)
The name of the RFC Function Component registered with IBM Tivoli Directory Integrator. This option should be changed only on the advice of IBM support. The default value is:

```
ibmdi.SapR3RfcFC
```

Add Mode StyleSheets (sapr3.userconn.putStylesheets)
The list of XSLT style sheets files to be executed by the Connector when deployed in **Add Only** mode. At runtime, each style sheet is applied to the XML contained within the Container Entry. The XSL will be applied to the value of the attribute named **sapUserXml**. Each XSL style sheet filename must be entered on a new line within the text box. This configuration parameter should be changed only at the direction of IBM support. The default value is:

```
xsl/bapi_user_create.xsl, xsl/bapi_user_actgroups_assign.xsl,
   xsl/bapi_user_profiles_assign.xsl
```

Update Mode StyleSheets (sapr3.userconn.modifyStylesheets)
The list of XSLT style sheets files to be executed by the Connector when deployed in **Update** mode. At runtime, each style sheet is applied to the XML contained within the Container Entry. The XSL will be applied to the value of the attribute named **sapUserXml**. Each XSL style sheet filename must be entered on a new line within the text box. This configuration parameter should be changed only at the direction of IBM support. The default XSL list is:

```
xsl/bapi_user_change.xsl, xsl/bapi_user_actgroups_assign.xsl,
   xsl/bapi_user_profiles_assign.xsl
```

Delete Mode StyleSheets (sapr3.userconn.deleteStylesheets)
The list of XSLT style sheets files to be executed by the Connector when deployed in **Delete** mode. At runtime, each style sheet is applied to the XML contained within the Container Entry. The XSL will be applied to the value of the attribute named **sapUserXml**. Each XSL style sheet filename must be entered on a new line within the text box. This configuration parameter should be changed only at the direction of IBM support. The default value is:

```
xsl/bapi_user_delete.xsl
```

Lookup Mode Pre StyleSheet (sapr3.userconn.findPreStylesheet)
The XSLT style sheet file to be executed by the Connector when creating an RFC XML request that is able to obtain all user attributes for a given user. This configuration value must be set when the Connector is deployed in **Update**, **Delete**, and **Lookup** modes. This configuration parameter should be changed only at the direction of IBM support. The default value is:

```
xsl/bapi_user_getdetail_precall.xsl
```

Lookup Mode Post StyleSheet (sapr3.userconn.findPostStylesheet)
The XSLT style sheet file to be executed by the Connector when creating the user XML formatted response from the Connector. This configuration value must be set when the Connector is deployed in **Update**, **Delete**, and **Lookup** modes. The XSLT transforms the response XML from the RFC executed as a result of the XSLT from **Lookup Mode Pre StyleSheet** configuration. This configuration parameter should be changed only at the direction of IBM support. The default value is:

```
xsl/bapi_user_getdetail_postcall.xsl
```

Select Entries Pre StyleSheet (sapr3.userconn.selectEntriesPreStylesheet)
The XSLT style sheet file to be executed by the Connector when creating an RFC XML request that is able to obtain all user names from SAP. This configuration value must be set when the Connector is deployed in **Iterator** mode. This configuration parameter should be changed only at the direction of IBM support. The default value is:

```
xsl/bapi_user_getlist_precall.xsl
```

Select Entries Post StyleSheet (sapr3.userconn.selectEntriesPostStylesheet)
The XSLT style sheet file to be executed by the Connector when creating the user XML for the **getNextEntry()** processing. This configuration value must be set when the Connector is deployed in **Iterator** mode. The XSLT transforms the response XML from the RFC executed as a result of the XSLT from **Select Entries Pre StyleSheet** configuration. This configuration parameter should be changed only at the direction of IBM support. The default value is:

```
xsl/bapi_user_getlist_postcall.xsl
```

Iterator Mode Pre StyleSheet (sapr3.userconn.getNextPreStylesheet)
The XSLT style sheet file to be executed by the Connector when creating an RFC XML request that is able to obtain all user attributes for a given user. This configuration value must be set when the Connector is deployed in **Iterator** mode. This configuration parameter should be changed only at the direction of IBM support. The default value is:

```
xsl/bapi_user_getdetail_precall.xsl
```

Iterator Mode Post StyleSheet (sapr3.userconn.getNextPostStylesheet)
The XSLT style sheet file to be executed by the Connector when creating the user XML formatted response from the Connector. This configuration value must be set when the Connector is deployed in **Iterator** mode. The XSLT transforms the response XML from the RFC executed as a result of the XSLT from **Iterator Mode Pre StyleSheet** configuration. This configuration parameter should be changed only at the direction of IBM support. The default value is:

```
xsl/bapi_user_getdetail_postcall.xsl
```

Detailed Log
When checked, generates additional log messages. The Connector logs data and activity when this option is enabled.

Using the User Registry Connector for SAP ABAP Application Server

This section describes how to use the Connector in each of the IBM Tivoli Directory Integrator Connector modes. The section also describes the IBM Tivoli Directory Integrator Entry schema supported by the Connector.

Note: The default XSL style sheet file name values are relative path locations with respect to the IBM Tivoli Directory Integrator AssemblyLine execution directory. In some situations, it may be necessary to preprend the default file name values with the fully qualified installation location of the XSL files. Such modification is likely if the IBM Tivoli Directory Integrator Component Suite for SAP ABAP Application Server has been installed in (or if the AssemblyLine is executing from) a directory location separate from the IBM Tivoli Directory Integrator installation.

IBM Tivoli Directory Integrator Entry Schema

The User Registry Connector supports only two fixed IBM Tivoli Directory Integrator entry attributes. The schema is available through the **discover schema** feature (the torch icon) in the IBM Tivoli Directory Integrator configuration tool. The attribute schema is described below.

Table 68. IBM Tivoli Directory Integrator Schema

Attribute Name	Type	Description
sapUserXml	java.lang.String	A string representing the attributes of an SAP user. The XSchema is defined in "XSchema for User Registry Connector XML" on page 487. This attribute and value must be present on the **Output Map** when the Connector is deployed in **Add Only**, **Update** and **Delete** modes. This attribute and value are available on the **Input Map** when the Connector is deployed in **Lookup** and **Iterator** modes.

Table 68. IBM Tivoli Directory Integrator Schema (continued)

Attribute Name	Type	Description
sapUserName	java.lang.String	A string representing the name of a given SAP user. The Connector supports this attribute primarily for configuration of **Link Criteria**.

Add Only Mode

When deployed in **Add Only** mode, the Connector is able to create a new user in the SAP database. The Connector should be added to the **Flow** section of a IBM Tivoli Directory Integrator AssemblyLine. The **Output Map** must define a mapping for the **sapUserXml** Connector attribute. The value of this attribute represents the details of the user to be added to SAP. The value will be applied to each configured XSLT file in the order defined. The XSLT transforms produce separate RFC XML requests to be executed by the RFC Function Component, which is managed internally by the Connector.

The Connector does not support duplicate or multiple entries. Only one entry should be supplied to the Connector at a time.

Update Mode

When deployed in **Update** mode, the Connector is able to modify an existing user in the SAP database. The Connector should be added to the **Flow** section of a IBM Tivoli Directory Integrator AssemblyLine. The **Output Map** must define a mapping for the **sapUserXml** Connector attribute. The value of this attribute represents the details of the user to be changed in SAP. The value will be applied to each configured XSLT file in the order defined. The XSLT transforms produce separate RFC XML requests to be executed by the RFC Function Component, which is managed internally by the Connector.

Additionally, the **sapUserName** attribute should be defined in the **Link Criteria** of the Connector. The **Link Criteria** is required by the AssemblyLine, since the AssemblyLine will invoke the Connectors **findEntry()** method to verify the existence of the given user. The value of **sapUserName**, as defined in the **Link Criteria**, must match the value of the `<sapUserName>` XML element present in the value of **sapUserXml**. All parameters defined in the **Link Criteria** are passed as XSLT style sheet parameters. If duplicate **Link Criteria** names are supplied, the Connector will use the last value supplied. The style sheets are not required to use the parameter.

The only operator supported for **Link Criteria** is an **equals exact match**. Wildcard search criteria are not supported, because the RFC lookup method does not currently support wild cards. The Connector will not return duplicate entries.

The Connector does not support duplicate or multiple entries. Only one entry should be supplied to the Connector at a time.

Note: This mode allows role and profile assignments to be changed. If **sapRoleList** or **sapProfileList** are present in the XML supplied to the Connector, then Connector will perform a complete delete and replace of the current assignments in SAP. This means the supplied XML must contain the complete assignments that need to exist after the operation is executed. This is true also for date ranges associated with roles. If the intention is to change a date range for a role already assigned, and not add or remove existing assignments, the complete list of role assignments with the new date ranges needs to be supplied in the XML. Date ranges should be present with all roles, unless the SAP defaults date values are acceptable.

Delete Mode

When deployed in **Delete** mode, the Connector is able to delete an existing user from the SAP database. The Connector should be added to the **Flow** section of a IBM Tivoli Directory Integrator AssemblyLine. The **sapUserName** attribute must be defined in the **Link Criteria** of the Connector. The **Link Criteria** is required by the AssemblyLine, since the AssemblyLine will invoke the Connector's **findEntry()** method to verify the existence of the given user. All parameters defined in the **Link Criteria** are passed as XSLT

style sheet parameters. If duplicate **Link Criteria** names are supplied, the Connector will use the last value supplied. The style sheets are not required to use the parameter.

The only operator supported for **Link Criteria** is an equals exact match. Wildcard search criteria are not supported, because the RFC lookup method does not currently support wild cards.

The Connector does not support duplicate or multiple entries. Only one entry should be supplied to the Connector at a time.

Lookup Mode

When deployed in **Lookup** mode, the Connector is able to obtain all details of a given SAP user. The Connector should be added to the **Flow** section of a IBM Tivoli Directory Integrator AssemblyLine. The **sapUserName** attribute must be defined in the **Link Criteria** of the Connector. If duplicate **Link Criteria** names are supplied, the Connector will use the last value supplied. The Connector will populate the XML string value of the attribute **sapUserXml**. This attribute is available to the AssemblyLine in the Connector's **Input Map** .

The Connector's **findEntry()** method is the main code executed. It uses the result of the XSLT transform configured in **Lookup Mode Pre StyleSheet**, to execute an RFC to obtain all details for the given user. The result of the RFC is then transformed using the XSLT transform configured in **Lookup Mode Post StyleSheet**.

The only operator supported for **Link Criteria** is an equals exact match. Wildcard search criteria are not supported, because the RFC lookup method does not currently support wild cards.

The Connector does not support duplicate or multiple entries. The Connector will return only one entry at a time.

Iterator Mode

When deployed in **Iterator** mode, the Connector is able to retrieve the details of each user in the SAP database, in turn, and make those details available to the AssemblyLine. The XSLT style sheets for **Select Entries Pre StyleSheet**, **Select Entries Post StyleSheet**, **Iterator Mode Pre StyleSheet**, and **Iterator Mode Post StyleSheet** must be configured.

When deployed in this mode, the IBM Tivoli Directory Integrator AssemblyLine will first call the Connector's **selectEntries()** method to obtain and cache a list of all user names in the SAP database. The AssemblyLine will then call the Connector's **getNextEntry()** method. This method will maintain a pointer to the current name cached in the list. The method will use this name to call an RFC to obtain all details for the user. The results of the RFC are formatted by an XSLT transform and set as the value of **sapUserXml** and returned by the Connector.

Transactional Operations Not Supported

Neither the Connector nor IBM Tivoli Directory Integrator currently supports transactions with SAP ABAP Application Server. Some of the known consequences are explained in this section.

When the Connector is deployed in a mode that results in write operations with SAP (that is, **Add Only**, **Update** and**Delete**) it is possible for operations to be partially complete. This can occur if multiple XSL style sheets, which generate RFC requests, are required to complete the operation. If one of the earlier RFC requests fails, then RFC requests executed subsequently may fail as a result. The Connector attempts to perform all XSL transformations and resulting RFC invocations on a best effort basis.

Consider the **Add Only** case to create a user account in SAP. The first style sheet generates an RFC request for `BAPI_USER_CREATE`. The second style sheet generates an RFC request for `BAPI_USER_ACTGROUPS_ASSIGN`. The third style sheet generates an RFC request for `BAPI_USER_PROFILES_ASSIGN`. If the third request fails, then the user may be created without the assignment of profiles.

Another case exists when attempting to create a user that already exists in SAP. The first style sheet results in a call to BAPI_USER_CREATE. This invocation will result in an ABAP application level error return result (this is not the same as an API or infrastructure error). The Connector will log this. The Connector will then proceed with the subsequent style sheet and RFC invocations, which attempt to assign roles and profiles to the user. Since the user already exists, the role and profile assignments will succeed.

For the case explained above, should the Connector stop processing after the first RFC, or should the Connector continue with the role and profile assignments that the IBM Tivoli Directory Integrator user expected to exist for the newly created user? If the required behavior is to stop after the first RFC error, then an additional configuration of the IBM Tivoli Directory Integrator AssemblyLine can satisfy this requirement. Deploy a second instance of the Connector in **Lookup** mode before the **Add Only** mode instance. The **Lookup** Connector can assist some custom JavaScript code to conditionally terminate or continue the AssemblyLine, depending on the existence of the user to be created.

Handling ABAP Errors

The Connector invokes BAPI/RFC functions in SAP to perform the Connector mode operations. In some cases, data passed to the BAPI/RFC functions from the XML input, may result in ABAP data validation failures. An example of this case could be the value for post code is not valid within the country region. The BAPI/RFC functions return the results of validation checks in the RETURN parameter of the RFC.

The Connector has been designed to make the RFC return status available to the AssemblyLine. The Connector does not interpret or translate ABAP errors or warnings into thrown exceptions. The Connector registers a script bean named **urcAbapErrorCache**. The bean is registered for all Connector modes and can be accessed in Connector hooks. The bean is an instance of **AbapErrorCache**. Script code in a Connector hook can use this information to perform contingency actions as required. The cache is reset before the execution of each Connector method.

Example script code is shown below. For specific details, refer to the Javadoc contained in the distribution package.

```
var errs = urcAbapErrorCache.getLastErrorSet();
if (errs.size() > 0) {
  task.logmsg("********** There were ABAP Errors **********");
  for (var i = 0; i < errs.size(); ++i) {
   var errInfo = errs.get(i);
   task.logmsg("The message is: " + errInfo.getMsg());
   task.logmsg("The message number is: " + errInfo.getMsgNum().toString());
  }
}

var warns = urcAbapErrorCache.getLastWarningSet();
if (warns.size() > 0) {
  task.logmsg("********** There were ABAP Warnings *********");
  for (var i = 0; i < warns.size(); ++i) {
   var errInfo = warns.get(i);
   task.logmsg("The message is: " + errInfo.getMsg());
   task.logmsg("The message number is: " + errInfo.getMsgNum().toString());
  }
}
```

Human Resources/Business Object Repository Connector for SAP ABAP Application Server

This section describes the configuration and operation of the IBM Tivoli Directory Integrator Human Resources/Business Object Repository Connector for SAP ABAP Application Server.

This chapter contains the following sections:

- "Introduction"
- "Configuration" on page 464
- "Using the Human Resources Connector for SAP ABAP Application Server" on page 466

Introduction

The SAP Human Resources modules include a large range of business features. The major feature areas address the business needs of payroll, personnel time management, and general personnel master data management.

From a data perspective, the backbone of the SAP HR system is the *infotype*. Infotypes are a logical grouping of related attributes. SAP defines a large set of default infotypes, which are grouped and identified in SAP using number ranges. The table below shows the ranges:

Table 69. Infotype Number Ranges

Number Range	HR Submodule
0000 to 0999	HR Master Data
1000 to 1999	Personnel Planning
2000 to 2999	Time Management
4000 to 4999	Recruitment
9000 to 9999	Custom extensions

Since there are such a large number of infotypes, it is quite difficult to design a single Tivoli Directory Integrator Connector to cover and suit all SAP HR integration requirements. Fortunately, SAP supports access to its HR data repositories via Business APIs (BAPI) that are attached to objects in the Business Object Repository (BOR). As a result, a generic BOR Connector has been designed and implemented. This Connector can invoke any method of any BOR object. The Connector projects an XML representation of the data managed by the given BOR object. The Connector requires the configuration of a set of XSL style sheets, and specification of the class identification name for the given BOR object (in fact, the XSL style sheets define the XML data representation).

The figure below illustrates the component design of the Connector.

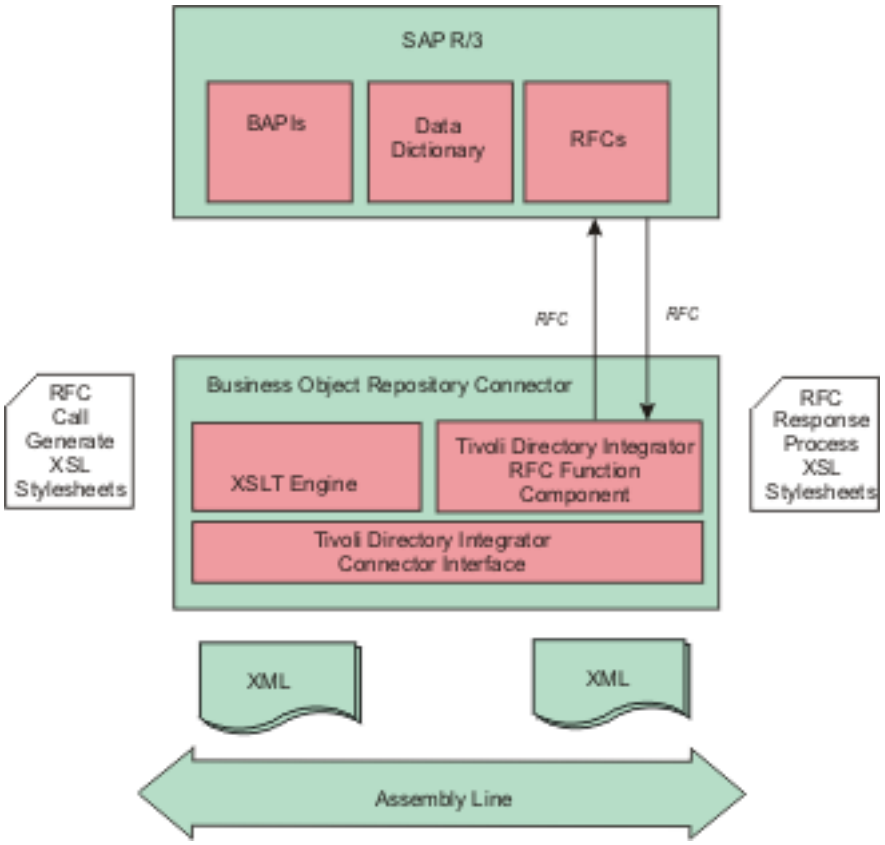


Figure 3. Component design of the Human Resources/Business Object Repository Connector for SAP ABAP Application Server

Tivoli Directory Integrator supplies an example set of XSL style sheets that enable the Connector to manage HR Personal Data (Infotype 0002). The style sheets have been setup to invoke the BAPI RFC methods of the `PERSDATA` BOR object. The Connector uses the generic RFC invocation feature of the Tivoli Directory Integrator Function Component for SAP ABAP Application Server.

The key features and benefits of the Connector are:

- Support for Create, Read, Update, and Delete (C.R.U.D) operations for SAP HR data.
- Modifiable behavior through XSL transformations for SAP ABAP Application Server RFC execution.
- Minimal compile time dependency between the Connector and SAP ABAP Application Server. The Connector does not use any generated RFC proxy code. It relies on the RFC Function Component as a dynamic proxy.
- No need for custom ABAP or Java coding (although specific new features might be supported with custom code).

The Connector supports the following standard Tivoli Directory Integrator Connector modes, but relies on the standard BAPI methods to deliver the functionality of each mode:

- Add Only
- Update
- Delete
- Lookup
- Iterator

The table below gives an example of Connector mode to BAPI method mappings

Table 70. Example Mappings

Connector Mode	BAPI Method
Add	Create, CreateFromData
Update	Change
Delete	Delete
Lookup	Get, GetDetail
Iterator	GetList, Get, GetDetailedList

Key Fields and XML Representation

Key fields of BOR objects are given special treatment by the Connector. This is reflected in the XML representation of BOR object data.

While it is possible to define alternate XSL style sheets to process request and response XML, the style sheets must support an element named **sapBorObjIdentifier**. This element is processed by the Java code of the Connector when returning entries in **Lookup** and **Iterator** modes. The **sapBorObjIdentifier** may appear anywhere within the XML. The contents of the element are elements whose tag names match the names of the key fields of the given BOR object.

The general form of the HR Personal Data XML is shown below.

```
<sapPersonalData>
 <sapBorObjIdentifier>
  <EmployeeNumber>00000001</EmployeeNumber>
  <SubType />
  <ObjectID />
  <LockIndicator />
  <ValidityEnd>99991231</ValidityEnd>
  <ValidityBegin>19740320</ValidityBegin>
  <RecordNumber>000</RecordNumber>
 </sapBorObjIdentifier>
 <personalDataDetail>
  <title>1</title>
  <firstname></firstname>
  <lastname></lastname>
  <nameAtBirth />
  <knownAs></knownAs>
  <surnamePrefix />
  <gender></gender>
  <dateOfBirth></dateOfBirth>
  <birthPlace />
  <stateOfBirth />
  <countryOfBirth />
  <maritalStatus></maritalStatus>
  <numberOfChildren></numberOfChildren>
  <religion />
  <language></language>
  <languageCode></languageCode>
  <nationality></nationality>
  <idNumber />
 </personalDataDetail>
</sapPersonalData>
```

Skip Lookup in Update and Delete mode

The Human Resources/Business Object Repository Connector for SAP ABAP Application Server supports the **Skip Lookup** general option in Update or Delete mode. When it is selected, no search is performed prior to actual update and delete operations.

For this to function, for HR Personal Data (infotype 0002), the following attributes must be defined in the Link Criteria:

- EmployeeNumber
- ValidityBegin
- ValidityEnd

Configuration

The BOR Connector for SAP ABAP Application Server (SAP ABAP AS) may be added directly into an assembly line. The following section lists the configuration parameters that are available for SAP client connections and XSL style sheet behavior. Runtime names are shown in parentheses.

Parameters

ABAP AS Client (client)
SAP ABAP AS Logon client for SAP connection (for example, *100*). This is passed directly to the RFC Function Component.

ABAP AS User (user)
SAP ABAP AS Logon user for SAP connection. This is passed directly to the RFC Function Component.

Password (passwd)
SAP ABAP AS Logon password for SAP connection. This is passed directly to the RFC Function Component.

ABAP AS System Number (sysnr)
The SAP ABAP AS system number for SAP connection (for example, *100*). This is passed directly to the RFC Function Component.

ABAP AS Hostname (ashost)
SAP ABAP AS application server name for SAP connection. This is passed directly to the RFC Function Component.

Gateway host (gwhost)
Gateway host name for SAP connection. This is passed directly to the RFC Function Component.

RFC Trace (trace)
Set to one (*1*) to enable RFC API tracing. If enabled, the SAP RFC API will produce separate `rfc_nnnn.trc` files in the working directory of Tivoli Directory Integrator. This option may be useful to help diagnose RFC invocation problems. It logs the activity and data between the Connector and SAP ABAP AS. This should be set to zero (*0*) for production deployment.

Optional RFC Connection Parameters
Used to define a list of other optional RFC connection parameters. The value for this configuration list is a key=value list where each connection parameter is separated by the space character. For example the following string value would set the SAP Gateway Service to "sapgw00" and enable the SAP GUI.

```
"gwserv=sapgw00 use_sapgui=1"
```

Here is a list of optional SAP Java Connector parameters that are accessible.

Alias user name (alias_user)
SAP message server (mshost)
Gateway service (gwserv)
Logon language (lang)
1 (Enable) or 0 (disable) RFC trace (trace)
Initial codepage in SAP notation (codepage)
Secure network connection (SNC) mode, 0 (off) or 1 (on) (snc_mode)
SNC partner, for example, p:CN=R3, O=XYZ-INC, C=EN (snc_partnername)

SNC level of security, 1 to 9 (snc_qop).
SNC name. Overrides default SNC partner (snc_myname)
Path to library which provides SNC service (snc_lib)
SAP R/3 name (r3name)
Group of SAP application servers (group)
Program ID of external server program (tpname)
Host of external server program (tphost)
Type of remote host 2 = R/2, 3 = R/3, E = External (type)
Enable ABAP debugging 0 or 1 (abap_debug)
Use remote SAP graphical user interface (0/1/2) (use_sapgui)
Get/Don't get a SSO ticket after logon (1 or 0) (getsso2)
Use the specified SAP Cookie Version 2 as logon ticket (mysapsso2)
Use the specified X509 certificate as logon ticket (x509cert)
Enable/Disable logon check at open time, 1 (enable) or 0 (disable) (lcheck)
String defined for SAPLOGON on 32-bit Windows (saplogon_id)
Data for external authentication (PAS) (extiddata)
Type of external authentication (PAS) (extidtype)
Idle timeout (in seconds) for the connection after which it will be closed by R/3. Only positive values are allowed. (idle_timeout)
Enable (1) or Disable (0) dsr support (dsr)

BOR Class Name (sapr3.conn.borObjName)
The name of the BOR class with which this Connector will be integrating. The names of BOR classes are available using transaction BAPI in SAP. This value is used to obtain the keyfield names of the BOR object when a schema query is performed.

RFC Function Component Name (sapr3.conn.rfcFC)
The name of the RFC Function Component that is registered with Tivoli Directory Integrator. This option should be changed only on the advice of IBM support. The default value is:

```
ibmdi.SapR3RfcFC
```

Add Mode StyleSheets (sapr3.conn.putStylesheets)
The list of XSLT style sheets files to be executed by the Connector when deployed in **Add Only** mode. At runtime, each style sheet is applied to the XML contained within the Container Entry. The XSL will be applied to the value of the attribute named **sapXml**. Each XSL style sheet filename must be entered on a new line within the text box.

Update Mode StyleSheets (sapr3.conn.modifyStylesheets)
The list of XSLT style sheets files to be executed by the Connector when deployed in **Update** mode. At runtime, each style sheet is applied to the XML contained within the Container Entry. The XSL will be applied to the value of the attribute named **sapXml**. Each XSL style sheet filename must be entered on a new line within the text box.

Delete Mode StyleSheets (sapr3.conn.deleteStylesheets)
The list of XSLT style sheets files to be executed by the Connector when deployed in **Delete** mode. At runtime, each style sheet is applied to the XML contained within the Container Entry. The XSL will be applied to the value of the attribute named **sapXml**. Each XSL style sheet filename must be entered on a new line within the text box.

Lookup Mode Pre StyleSheet (sapr3.conn.findPreStylesheet)
The XSLT style sheet file to be executed by the Connector when creating an RFC XML request that is able to obtain all user attributes for a given user. This configuration value must be set when the Connector is deployed in **Update**, **Delete**, and **Lookup** modes.

Lookup Mode Post StyleSheet (sapr3.conn.findPostStylesheet)
The XSLT style sheet file to be executed by the Connector when creating the user XML formatted response from the Connector. This configuration value must be set when the Connector is deployed in **Update**, **Delete**, and **Lookup** modes. The XSLT transforms the response XML from the RFC executed as a result of the XSLT from **Lookup Mode Pre StyleSheet** configuration.

Select Entries Pre StyleSheet (sapr3.conn.selectEntriesPreStylesheet)
The XSLT style sheet file to be executed by the Connector when creating an RFC XML request that is able to obtain all user names from SAP. This configuration value must be set when the Connector is deployed in **Iterator** mode.

Select Entries Post StyleSheet (sapr3.conn.selectEntriesPostStylesheet)
The XSLT style sheet file to be executed by the Connector when creating the user XML for the **getNextEntry()** processing. This configuration value must be set when the Connector is deployed in **Iterator** mode. The XSLT transforms the response XML from the RFC executed as a result of the XSLT from **Select Entries Pre StyleSheet** configuration.

Iterator Mode Pre StyleSheet (sapr3.conn.getNextPreStylesheet)
The XSLT style sheet file to be executed by the Connector when creating an RFC XML request that is able to obtain all user attributes for a given user. This configuration value must be set when the Connector is deployed in **Iterator** mode.

Iterator Mode Post StyleSheet (sapr3.conn.getNextPostStylesheet)
The XSLT style sheet file to be executed by the Connector when creating the user XML formatted response from the Connector. This configuration value must be set when the Connector is deployed in **Iterator** mode. The XSLT transforms the response XML from the RFC that is executed as a result of the XSLT from **Iterator Mode Pre StyleSheet** configuration.

Detailed Log
When checked, generates additional log messages. The Connector logs data and activity when this option is enabled.

Using the Human Resources Connector for SAP ABAP Application Server

This section describes the details of using the Connector in each of the supported Tivoli Directory Integrator Connector modes. The section also describes the Tivoli Directory Integrator Entry schema supported by the Connector.

Note: The default XSL style sheet file name values are relative path locations with respect to the Tivoli Directory Integrator AssemblyLine execution directory. In some situations, it may be necessary to preprend the default file name values with the fully qualified installation location of the XSL files. Such modification is likely if the Tivoli Directory Integrator Component Suite for SAP ABAP Application Server has been installed in (or if the AssemblyLine is executing from) a directory location separate from the Tivoli Directory Integrator installation.

IBM Tivoli Directory Integrator Entry Schema

The BOR Connector supports one native attribute named **sapXml**. The value of **sapXml** is an XML string representing the attributes of a BOR object. Other attributes reflect the given BOR object key field names. They are supported to allow the definition of IBM Tivoli Directory Integrator **Link Criteria** when the Connector is deployed in **Lookup**, **Delete**, or **Update** modes.

The schema is available via the query schema feature in the IBM Tivoli Directory Integrator configuration tool. The attribute schema is described below.

Table 71. Entry Schema Attributes

Attribute Name	Type	Description
sapXml	java.lang.String	A string representing the attributes of an SAP BOR Object. This attribute and value must be present on the **Output Map** when the Connector is deployed in **Add Only** and **Update** modes. This attribute is available on the **Input Map** when the Connector is deployed in **Lookup** and **Iterator** modes.

Table 71. Entry Schema Attributes (continued)

Attribute Name	Type	Description
EmployeeNumber	java.lang.String	Personal Data Infotype 0002 specific. The 8 digit employee number. This attribute and value must be present on the **Link Criteria** when the Connector is deployed in **Lookup**, **Update** and **Delete** modes. This attribute is available on the **Input Map** when the Connector is deployed in **Lookup** and **Iterator** modes.
Subtype	java.lang.String	Personal Data Infotype 0002 specific. The 4 character personal data subtype. This attribute and value must be present on the **Link Criteria** when the Connector is deployed in **Lookup**, **Update** and **Delete** modes. This attribute is available on the **Input Map** when the Connector is deployed in **Lookup** and **Iterator** modes.
ObjectID	java.lang.String	Personal Data Infotype 0002 specific. The 2 character object ID for infotypes where all other key fields are the same. This attribute is available on the **Input Map** when the Connector is deployed in **Lookup** and **Iterator** modes.
LockIndicator	java.lang.String	Personal Data Infotype 0002 specific. The 1 character flag indicating if the master data record is locked in SAP. This attribute is available on the **Input Map** when the Connector is deployed in **Lookup** and **Iterator** modes.
ValidityEnd	java.lang.String	Personal Data Infotype 0002 specific. 8 digit date value (YYYYMMDD). This attribute and value must be present on the **Link Criteria** when the Connector is deployed in **Lookup**, **Update** and **Delete** modes. This attribute is available on the **Input Map** when the Connector is deployed in **Lookup** and **Iterator** modes.
ValidityBegin	java.lang.String	Personal Data Infotype 0002 specific. 8 digit date value (*YYYYMMDD*). This attribute and value must be present on the **Link Criteria** when the Connector is deployed in **Lookup**, **Update** and **Delete** modes. This attribute is available on the **Input Map** when the Connector is deployed in **Lookup** and **Iterator** modes.
RecordNumber	java.lang.String	Personal Data Infotype 0002 specific. 2 digit value. This attribute is available on the **Input Map** when the Connector is deployed in **Lookup** and **Iterator** modes.

Add Only Mode

When deployed in **Add Only** mode, the Connector is able to create a new object in the SAP database. The Connector should be added to the **Flow** section of a Tivoli Directory Integrator AssemblyLine. The **Output Map** must define a mapping for the **sapXml** Connector attribute. The value of this attribute represents the details of the object to be added to SAP. The value will be applied to each configured XSLT file in the order defined. The XSLT transforms produce separate RFC XML requests to be executed by the RFC Function Component, which is managed internally by the Connector.

The Connector does not support duplicate or multiple entries. Only one entry should be supplied to the Connector at a time.

For HR Personal Data (infotype 0002), a valid employee number must exist. The general form of the XML is shown below. The mandatory elements are **EmployeeNumber**, **ValidityBegin**, and **ValidityEnd**.

```
<sapPersonalData>
 <sapBorObjIdentifier>
  <EmployeeNumber>00000001</EmployeeNumber>
  <SubType />
  <ObjectID />
  <LockIndicator />
  <ValidityEnd>99991231</ValidityEnd>
  <ValidityBegin>19740320</ValidityBegin>
  <RecordNumber>000</RecordNumber>
 </sapBorObjIdentifier>
 <personalDataDetail>
  <title></title>
  <firstname></firstname>
  <lastname></lastname>
  <nameAtBirth />
  <knownAs>Torpedo</knownAs>
  <surnamePrefix />
  <gender>1</gender>
  <dateOfBirth></dateOfBirth>
  <birthPlace />
  <stateOfBirth />
  <countryOfBirth />
  <maritalStatus></maritalStatus>
  <numberOfChildren></numberOfChildren>
  <religion />
  <language></language>
  <languageCode></languageCode>
  <nationality></nationality>
  <idNumber />
 </personalDataDetail>
</sapPersonalData>
```

Update Mode

When deployed in **Update** mode, the Connector is able to modify an existing object in the SAP database. The Connector should be added to the **Flow** section of a Tivoli Directory Integrator AssemblyLine. The **Output Map** must define a mapping for the **sapXml** Connector attribute. The value of this attribute represents the details of the user to be changed in SAP. The value will be applied to each configured XSLT file in the order defined. The XSLT transforms produce separate RFC XML requests to be executed by the RFC Function Component, which is managed internally by the Connector.

Additionally, some of the key fields of the BOR object are needed for the **Link Criteria** of the Connector. The **Link Criteria** is required by the AssemblyLine, since the AssemblyLine will invoke the Connector's **findEntry()** method to verify the existence of the given object. All parameters defined in the **Link Criteria** are passed as XSLT style sheet parameters. If duplicate **Link Criteria** names are supplied, the Connector will use the last value supplied.

The Connector does not support duplicate or multiple entries. Only one entry should be supplied to the Connector at a time.

For HR Personal Data (infotype 0002), the following attributes must be defined in the **Link Criteria**:

- EmployeeNumber,
- ValidityBegin,
- ValidityEnd.

Since these attributes are passed as parameters to the XSL style sheets, they are not required in the XML. The general form of the XML is shown below.

```
<sapPersonalData>
 <sapBorObjIdentifier>
  <SubType />
```

```
  <ObjectID />
  <LockIndicator />
  <RecordNumber>000</RecordNumber>
 </sapBorObjIdentifier>
 <personalDataDetail>
  <title></title>
  <firstname></firstname>
  <lastname></lastname>
  <nameAtBirth />
  <knownAs>Torpedo</knownAs>
  <surnamePrefix />
  <gender></gender>
  <dateOfBirth></dateOfBirth>
  <birthPlace />
  <stateOfBirth />
  <countryOfBirth />
  <maritalStatus></maritalStatus>
  <numberOfChildren></numberOfChildren>
  <religion />
  <language></language>
  <languageCode></languageCode>
  <nationality></nationality>
  <idNumber />
 </personalDataDetail>
</sapPersonalData>
```

Delete Mode

When deployed in **Delete** mode, the Connector is able to delete an existing object from the SAP database. The Connector should be added to the **Flow** section of a Tivoli Directory Integrator AssemblyLine. In **Delete** mode, the Connector relies solely on the **Link Criteria**. All parameters defined in the **Link Criteria** are passed as XSLT style sheet parameters. If duplicate **Link Criteria** names are supplied, the Connector will use the last value supplied.

The Connector does not support duplicate or multiple entries. Only one entry should be supplied to the Connector at a time.

For HR Personal Data (infotype 0002), the following Attributes must be defined in the **Link Criteria**:

- EmployeeNumber,
- ValidityBegin,
- ValidityEnd.

Lookup Mode

When deployed in **Lookup** mode, the Connector is able to obtain all details of a given SAP object. The Connector should be added to the **Flow** section of a Tivoli Directory Integrator AssemblyLine. Connector key field attributes should be defined in the **Link Criteria** of the Connector. If duplicate **Link Criteria** names are supplied, the Connector will use the last value supplied. The Connector will populate the XML string value of the attribute **sapXml** and make it available to the AssemblyLine in the Connector's **Input Map**. The key field names and values are also made available to the **Input Map**.

The Connector's **findEntry()** method is the main code executed. It uses the result of the XSLT transform configured in **Lookup Mode Pre StyleSheet** to execute an RFC and obtain all details for the given user. The result of the RFC is then transformed using the XSLT transform configured in **Lookup Mode Post StyleSheet**.

The Connector does not support duplicate or multiple entries. The Connector will return only entry at a time.

For HR Personal Data (infotype 0002), the following Attributes must be defined in the **Link Criteria**:

- EmployeeNumber,

- ValidityBegin,
- ValidityEnd.

Iterator Mode

When deployed in **Iterator** mode, the Connector is able to retrieve the details of each object in the SAP database, in turn, and make those details available to the AssemblyLine. The XSLT style sheets for **Select Entries Pre StyleSheet**, **Select Entries Post StyleSheet**, **Iterator Mode Pre StyleSheet**, and **Iterator Mode Post StyleSheet** must be configured.

When deployed in this mode, the Tivoli Directory Integrator AssemblyLine will first call the Connector's **selectEntries()** method to obtain and cache a list of all key field names and values (for the given BOR object) in the SAP database. The AssemblyLine will then call the Connector's **getNextEntry()** method. This method will maintain a pointer to the current key field cached in the list. The method will use the key field information to call an RFC to obtain all details for the object. The result of the RFC are formatted by an XSLT transform and set as the value of **sapXml** and returned by the Connector. The key field names and values are also made available to the **Input Map**.

Transactional Operations Not Supported

When the Connector is deployed in a mode that results in write operations with SAP (**Add Only**, **Update**, **Delete**), it is possible for operations to be partially complete. This can occur if multiple XSL style sheets, which generate RFC requests, are required to complete the operation. If one of the earlier RFC requests fails, then RFC requests executed subsequently may fail as a result.

Handling ABAP Errors

The Connector invokes BAPI/RFC functions in SAP to perform the Connector mode operations. In some cases, data passed to the BAPI/RFC functions from the XML input, may result in ABAP data validation failures. The BAPI/RFC functions return the results of validation checks in the "RETURN" parameter of the RFC.

The Connector has been designed to make the RFC return status available to the AssemblyLine. The Connector does not interpret or translate ABAP errors or warnings into thrown exceptions. The Connector registers a script bean named **borcAbapErrorCache**. The bean is registered for all Connector modes and can be accessed in Connector hooks. The bean is an instance of **AbapErrorCache**. Script code in a Connector hook can use this information to perform contingency actions as required. The cache is reset before the execution of each Connector method.

Example script code is shown below. For specific details, refer to the Javadoc contained in the distribution package.

```
var errs = borcAbapErrorCache.getLastErrorSet();
if (errs.size() > 0) {
  task.logmsg("********** There were ABAP Errors *********");
  for (var i = 0; i < errs.size(); ++i) {
   var errInfo = errs.get(i);
   task.logmsg("The message is: " + errInfo.getMsg());
   task.logmsg("The message number is: " + errInfo.getMsgNum().toString());
  }
}

var warns = borcAbapErrorCache.getLastWarningSet();
if (warns.size() > 0) {
  task.logmsg("********** There were ABAP Warnings *********");
  for (var i = 0; i < warns.size(); ++i) {
   var errInfo = warns.get(i);
   task.logmsg("The message is: " + errInfo.getMsg());
   task.logmsg("The message number is: " + errInfo.getMsgNum().toString());
  }
}
```

ALE Intermediate Document (IDOC) Connector for SAP ABAP Application Server and SAP ERP

This section describes the configuration and operation of the IBM Tivoli Directory Integrator Connector for processing ALE IDOCs sent from an SAP ABAP Application Server or ERP system. The chapter contains the following sections:

- "Introduction"
- "Configuration" on page 472
- "Using the SAP ALE IDOC Connector" on page 474

Introduction

In an SAP System the Application Link Enabling (ALE) is one of the core integration technologies. It involves the exchange of hierarchical data documents known as Intermediate Documents (IDOCs). There are two scenarios, inbound to SAP, and outbound from SAP. This release of the connector only integrates with IDOCs that are outbound from SAP, and inbound to IBM Tivoli Directory Integrator. This document will use the term inbound with reference to inbound to IBM Tivoli Directory Integrator. The SAP System will always be the IDOC client with IBM Tivoli Directory Integrator acting as the IDOC Server. The IDOC is sent to IBM Tivoli Directory Integrator as an asynchronous event, and when received, IBM Tivoli Directory Integrator pushes the IDOC data onto an AssemblyLine for processing as desired. As it is asynchronous communication, IBM Tivoli Directory Integrator will not provide a response to the client SAP system. SAP TID management is used to ensure data consistency between the SAP system client and IBM Tivoli Directory Integrator. Due to the asynchronous communication the Connector supports only Iterator mode.

When configuring ALE in an SAP System, the core task is to create what is called a distribution model. There are many pre-defined distribution models available as standard in an SAP system, but there is also the ability to create a customizable distribution model of your own. The core use of this connector is as an external application that acts as a logical system within the chosen SAP distribution model. The examples provided in this chapter will define integration into a custom SAP HR distribution model, and the pre-defined SAP Central User Management (CUA) distribution model. Of course the connector could be used for integration into any of the other SAP distribution models to access the master data for other SAP modules such as SAP FI/CO or SAP PP. For detailed information on SAP modules visit the SAP help site at http://help.sap.com. Almost any SAP master data business object with an IDOC interface can be exchanged this way.

Central to creating or configuring an SPA distribution model are the IDOC message types you want to support. What the connector provides is an XML version of the IDOC, which must be parsed accordingly. To facilitate parsing of the IDOC XML data the connector has been enabled to make use of the IBM Tivoli Directory Integrator XML parsers. You have the choice of the DOM Parser, the SAX Parser or the XSLT Parser. Using these will enable you to extract the required data from the IDOC for you business purpose. For example you may wish to extract particular infotypes from the SAP HR IDOC message type HRMD_A to forward on to IBM Tivoli Identity Manager for automated provisioning purposes.

The figure below illustrates the interaction with the SAP System IDOC client and the IBM Tivoli Directory Integrator IDOC server.

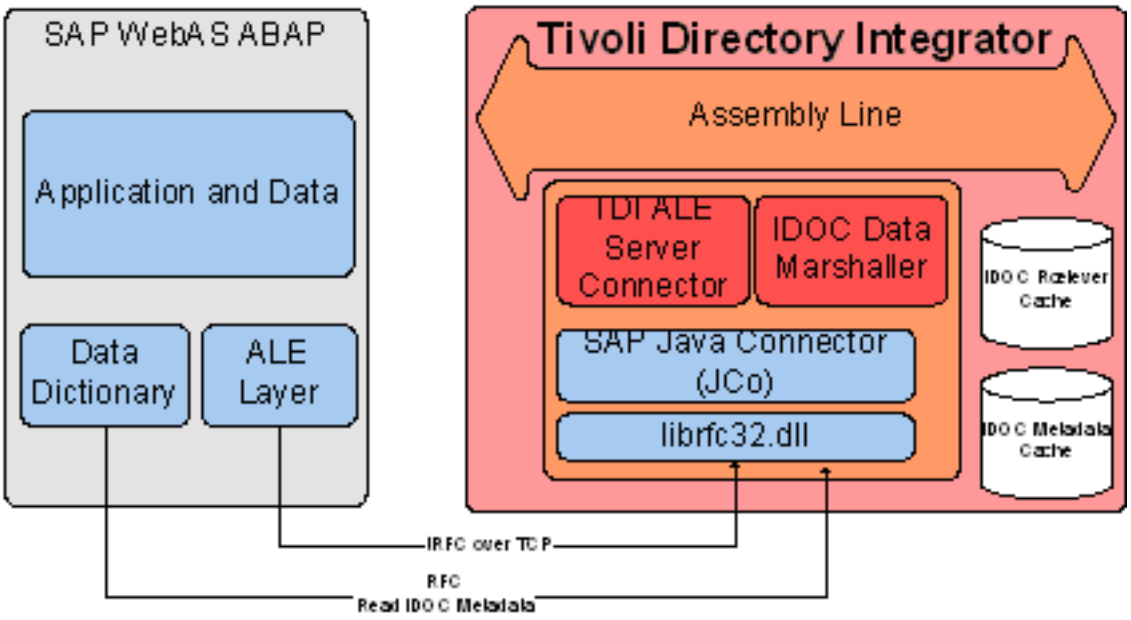


Figure 4. Interaction with the SAP System IDOC client and the IBM Tivoli Directory Integrator IDOC server

Installation

The SAP ALE IDOC Connector for SAP ABAP Application Server is part of the IBM Tivoli Directory Integrator 7.1 installation package. However, in order for the Connector to function correctly, some SAP class libraries need to be obtained and installed, alongside the `sapjco.jar` file as outlined in the installation instructions for the entire Component Suite:

- sapidoc.jar
- sapidocjco.jar

Configuration

The SAP ALE IDOC Connector for SAP ABAP Application Server may be added directly into an assembly line. The following section lists the configuration parameters the connector.

IDOC Server Parameters

IDOC Server SAP Gateway Host
A mandatory RFC Server Connection attribute that defines the host system that is the SAP Gateway.

IDOC Server SAP Gateway Service
A mandatory RFC Server Connection attribute that defines the SAP Gateway Service.

IDOC Server Program ID
A mandatory RFC Server Connection attribute that Defines the Server Program ID that is used in the configuration of the required TCP/IP RFC Destination to register the Server with the SAP Gateway.

IDOC Server Unicode Connection?
An optional RFC Server Connection attribute that defines the host system that is the SAP Gateway.

IDOC Server Optional Connection Parameters
An optional RFC Server Connection attribute used to define a list of other optional RFC connection parameters. The value for this configuration list is a key=value list where each

connection parameter is separated by the space character. For example the following string value would set the SAP System number to "00" and enable the RFC trace mechanism:

```
"jco.server.trace=1 jco.server.sysnr=00"
```

IDOC Client Configuration Parameters

IDOC Client Number
A mandatory RFC Client Connection attribute that defines the SAP Systems client. Consists of a 3 digit string value such as "000" or "100" defining the logon client.

IDOC Client User
A mandatory RFC Client Connection attribute that defines the SAP User Account logon id. Typically this would be a SAP User Account type of Communication or CPIC, although the Dialogue type can be used

IDOC Client Password
A mandatory RFC Client Connection attribute that defines the SAP User Account password.

IDOC Client Lang
A mandatory RFC Client Connection attribute that defines the logon language.

IDOC Client Hostname
A mandatory RFC Client Connection attribute that defines the hostname for the target SAP System.

IDOC Client System Number
A mandatory RFC Client Connection attribute that defines the system identifier (SID) for the target SAP System.

IDOC Client SAP Gateway Service
An optional RFC Client Connection attribute that defines the SAP Gateway Service. In most cases this will have the same value as the IDOC Server SAP Gateway Service.

IDOC Client SAP Gateway Host
An optional RFC Client Connection attribute that defines the SAP Gateway hostname.

IDOC Client Max Connections
A mandatory RFC Client Connection attribute that defines the maximum number of RFC connections supported by the internal connection pool.

IDOC Client Optional Connection Parameters
An optional RFC Client Connection attribute used to define a list of other optional RFC connection parameters. The value for this configuration list is a key=value list where each connection parameter is separated by the space character. For example the following string value would turn on the RFC trace mechanism and enable the use of the SAP GUI if it was installed on the same host as IBM Tivoli Directory Integrator.

```
"jco.client.trace=1 jco.client.use_sapgui=1"
```

The following is a list of optional SAP Java Connector parameters that are accessible.

Alias user name [jco.client.alias_user]
SAP message server [jco.client.mshost]
RFC trace [jco.client.trace]
Initial codepage in SAP notation [jco.client.codepage]
Secure network connection (SNC) mode, 0 (off) or 1 (on) [jco.client.snc_mode]
SNC partner, for example, p:CN=R3, O=XYZ-INC, C=EN [jco.client.snc_partnername]
SNC level of security, 1 to 9 [jco.client.snc_qop]
SNC name. Overrides default SNC partner [jco.client.snc_myname]
Path to library, which provides SNC service [jco.client.snc_lib]
SAP R/3 name [jco.client.r3name]
Group of SAP application servers [jco.client.group]

Program ID of external server program [jco.client.tpname]
Host of external server program [jco.client.tphost]
Type of remote host 2 = R/2, 3 = R/3, E = External [jco.client.type]
Enable ABAP debugging 0 or 1 [jco.client.abap_debug]
Use remote SAP graphical user interface (0/1/2) [jco.client.use_sapgui]
Get/Don't get a SSO ticket after logon (1 or 0) [jco.client.getsso2]
Use the specified SAP Cookie Version 2 as logon ticket [jco.client.mysapsso2]
Use the specified X509 certificate as logon ticket [jco.client.x509cert]
Enable/Disable logon check at open time, 1 (enable) or 0 (disable) [jco.client.lcheck]
String defined for SAPLOGON on 32-bit Windows [jco.client.saplogon_id]
Data for external authentication (PAS) [jco.client.extiddata]
Type of external authentication (PAS) [jco.client.extidtype]
Idle timeout (in seconds) for the connection after which it will be closed by R/3.
Only positive Enable (1) or Disable (0) dsr support [jco.client.dsr]

General Configuration Parameters

IDOC As XML Only?

A general attribute that defines if only the XML valued attribute for the IDOC is required. If set to "No", the IDOC control data will be set as independent attributes within the resulting Entry for each IDOC. If set to "Yes", only one attribute (idoc.xml) is created for the IDOC which contains the IDOC content as an XML valued string.

Process SAP RFM Requests?

A general attribute that defines if the Connector will also process remote function module (RFM) calls made on it. If set to "Yes" all RFM calls will be added to an Entry which contain one attribute (rfm.xml) which is the content of the RFM as an XML valued string. If set to "No" then RFM requests on the IDOC Server will be ignored.

Note: The processing performed for RFM calls is merely to provide the RFM as an XML valued attribute. The IDOC Server does not currently attempt to populate the export and table arguments of the RFM call. If required this can be provided under an enhancement of the Connector. To do this contact IBM Support. Only RFM calls that form part of an ALE distribution models internal process should be considered.

Parse IDOC or RFM XML?

A general attribute that defines if parsing is to be attempted on the XML valued attributes idoc.xml and rfm.xml. You must have one of the available IBM Tivoli Directory Integrator parsers configured to interact with the Connector in your AssemblyLine configuration.

Enable JCo Middleware Trace Logging?

A general attribute that defines if the available JCo trace logs are to be enabled and included in the AssemblyLine logging and tracing.

JCo Middleware Trace Level

A general attribute that defines the JCo middleware trace level.

JCo Middleware Trace File Path

A general attribute that defines the directory where the JCo middleware trace file will be created. Also used to store RFM requests as a file with XML content.

Using the SAP ALE IDOC Connector

This section describes details on using the Connector in the supported Iterator mode. Also described is the IBM Tivoli Directory Integrator schema supported by the Connector.

IBM Tivoli Directory Integrator schema

The schema for the Connector is centred on providing an AssemblyLine with Entries, where each represents and individual IDOC. An IDOC itself contains 3 sections of data. These are Control Data, Segment Data, and Status Data. The simplest and most effective way of representing this data in IBM

Tivoli Directory Integrator is an XML format, which can be easily dissected for the required data. As the control data is readily accessible, and can provide useful standalone information, this data is also available as individual attributes. The configuration parameter "IDOC As XML Only?" is used to enable or disable the production of the control data as stand alone attributes.

As the Connector is also able to accept Remote Function Module requests, there is a requirement to represent the data in one or more attributes. Currently the content of an RFM will be available as a single XML valued attribute. The configuration parameter "Process SAP RFM Requests?" is used to enable or disable the production of the RFM XML valued attribute.

The table below defines the schema available to this Connector.

Table 72. SAP ALE IDOC Connector Schema

Attribute Name	Attribute Description	Attribute Syntax
idoc.tid	Input schema attribute whose value is the associated TID value provided by the SAP System Client.	java.lang.string
idoc.xml	Input schema attribute whose value is the complete IDOC in XML format.	java.lang.string
idoc.segments.xml	Input schema attribute whose value is the complete Segment hierarchy in XML format. No control attribute values are contained in this XML.	java.lang.string
idoc.ctrl.ArchiveKey	Input schema attribute whose value represents the IDOC control data archive key (the value of the field "ARCKEY").	java.lang.string
idoc.ctrl.Client	Input schema attribute whose value represents the IDOC control data client (the value of the field "MANDT").	java.lang.string
idoc.ctrl.CreationDate	Input schema attribute whose value represents the IDOC control data creation date (the value of the field "CREDAT").	java.lang.string
idoc.ctrl.CreationTime	Input schema attribute whose value represents the IDOC control data creation time (the value of the field "CRETIM").	java.lang.string
idoc.ctrl.Direction	Input schema attribute whose value represents the IDOC control data direction (the value of the field "DIRECT").	java.lang.string
idoc.ctrl.EDIMessage	Input schema attribute whose value represents the IDOC control data EDI message (the value of the field "REFMES").	java.lang.string
idoc.ctrl.EDIMessageGroup	Input schema attribute whose value represents the IDOC control data EDI message group (the value of the field "REFGRP").	java.lang.string
idoc.ctrl.EDIMessageType	Input schema attribute whose value represents the IDOC control data EDI message type (the value of the field "STDMES").	java.lang.string
idoc.ctrl.EDIStandardFlag	Input schema attribute whose value represents the IDOC control data EDI standard flag (the value of the field "STD").	java.lang.string
idoc.ctrl.EDIStandardVersion	Input schema attribute whose value represents the IDOC control data EDI standard version (the value of the field "STDVRS").	java.lang.string
idoc.ctrl.EDITransmissionFile	Input schema attribute whose value represents the IDOC control data EDI transmission file (the value of the field "REFINT").	java.lang.string
idoc.ctrl.ExpressFlag	Input schema attribute whose value represents the IDOC control data express flag (the value of the field "EXPRSS").	java.lang.string
idoc.ctrl.IDocCompoundType	Input schema attribute whose value represents the IDOC control data IDOC compound type (the value of the field "DOCTYP").	java.lang.string

Table 72. SAP ALE IDOC Connector Schema (continued)

Attribute Name	Attribute Description	Attribute Syntax
idoc.ctrl.IDocNumber	Input schema attribute whose value represents the IDOC control data IDOC number (the value of the field "DOCNUM").	java.lang.string
idoc.ctrl.IDocSAPRelease	Input schema attribute whose value represents the IDOC control data IDOC SAP release (the value of the field "DOCREL").	java.lang.string
idoc.ctrl.IDocType	Input schema attribute whose value represents the IDOC control data IDOC type (the value of the field "IDOCTYP").	java.lang.string
idoc.ctrl.IDocTypeExtension	Input schema attribute who's value represents the IDOC control data IDOC type extension that is also known as CIM type or customer extension type (the value of the field "CIMTYP");	java.lang.string
idoc.ctrl.MessageCode	Input schema attribute whose value represents the IDOC control data message code (the value of the field "MESCOD").	java.lang.string
idoc.ctrl.MessageFunction	Input schema attribute whose value represents the IDOC control data message function (the value of the field "MESFCT").	java.lang.string
idoc.ctrl.MessageType	Input schema attribute whose value represents the IDOC control data message type (the value of the field "MESTYP").	java.lang.string
idoc.ctrl.OutputMode	Input schema attribute whose value represents the IDOC control data output mode (the value of the field "OUTMOD").	java.lang.string
idoc.ctrl.RecipientAddress	Input schema attribute whose value represents the IDOC control data recipient address (the value of the field "RCVSAD").	java.lang.string
idoc.ctrl.RecipientLogicalAddress	Input schema attribute whose value represents the IDOC control data logical recipient address (the value of the field "RCVLAD").	java.lang.string
idoc.ctrl.RecipientPartnerFunction	Input schema attribute whose value represents the IDOC control data recipient partner function (the value of the field "RCVPFC").	java.lang.string
idoc.ctrl.RecipientPartnerNumber	Input schema attribute whose value represents the IDOC control data recipient partner number (the value of the field "RCVPRN").	java.lang.string
idoc.ctrl.RecipientPartnerType	Input schema attribute whose value represents the IDOC control data recipient partner type (the value of the field "RCVPRT").	java.lang.string
idoc.ctrl.RecipientPort	Input schema attribute whose value represents the IDOC control data recipient port (the value of the field "RCVPOR").	java.lang.string
idoc.ctrl.SenderAddress	Input schema attribute whose value represents the IDOC control data sender address (the value of the field "SNDSAD").	java.lang.string
idoc.ctrl.SenderLogicalAddress	Input schema attribute whose value represents the IDOC control data logical sender address (the value of the field "SNDLAD").	java.lang.string
idoc.ctrl.SenderPartnerFunction	Input schema attribute whose value represents the IDOC control data sender partner function (the value of the field "SNDPFC").	java.lang.string
idoc.ctrl.SenderPartnerNumber	Input schema attribute whose value represents the IDOC control data sender partner number (the value of the field "SNDPRN").	java.lang.string

Table 72. SAP ALE IDOC Connector Schema (continued)

Attribute Name	Attribute Description	Attribute Syntax
idoc.ctrl.SenderPartnerType	Input schema attribute whose value represents the IDOC control data sender partner type (the value of the field "SNDPRT").	java.lang.string
idoc.ctrl.SenderPort	Input schema attribute whose value represents the IDOC control data Returns the sender port (the value of the field "SNDPOR").	java.lang.string
idoc.ctrl.Serialization	Input schema attribute whose value represents the IDOC control data serialization (the value of the field "SERIAL").	java.lang.string
idoc.ctrl.Status	Input schema attribute whose value represents the IDOC control data status (the value of the field "STATUS").	java.lang.string
idoc.ctrl.TableStructureName	Output schema attribute whose value represents the IDOC control data table structure name (the value of the field "TABNAM").	java.lang.string
idoc.ctrl.TestFlag	Input schema attribute whose value represents the IDOC control data test flag (the value of the field "TEST").	java.lang.string
rfm.xml	Input schema attribute whose value represents the complete content of an RMF request in XML format.	java.lang.string

Attributes of type `java.lang.String` can be of arbitrary length.

XML Attribute Parsing

The main mode of operation for the connector is the production of the XML valued attributes that represent the complete content of an IDOC, or RFM. As a result the best way to handle this data is with one of the available IBM Tivoli Directory Integrator XML Parsers attached to the Connector. Due to the nested nature of the resulting XML, the DOM parser is not recommended, but still can be used. The recommended parsers are the SAX Parser or the XSLT Parser depending on the type of SAP ALE distribution model the Connector is to integrate with. If the Connector has to handle multiple IDOC message types, or is configured to process RFM requests, then the SAX Parser is recommended. This is because you will have different XML schemas for the different IDOC message types, and the RFM XML. The SAX Parser is the only IBM Tivoli Directory Integrator parser that can handle XML values with different XML schemas. You do this by not configuring the SAX Parser to have a specific value for it's "Group" configuration parameter. This has the effect of not having to define a particular root element. If you are certain that the Connector will process only one type of IDOC, then you can use the XSLT Parser, which allows for a more complete Connector Entry to Work Entry attribute mapping. For example if the Connector was configured to be the recipient of SAP HR Master Data, then you would only ever expect to see IDOCs of the message type HRMD_A. At the time this connector was developed the latest version of this message type was HRMD_A06. You could then use and XSL like the following to parse the IDOCs contents for the required data.

```
<XSL:stylesheet xmlns:XSL="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <XSL:output method="XML" indent="yes" />

  <XSL:template match="HRMD_A06">
    <DocRoot>
      <Entry>
        <XSL:apply-templates select="./IDOC"/>
      </Entry>
    </DocRoot>
  </XSL:template>

  <XSL:template match="IDOC">
    <XSL:apply-templates select="./EDI_DC40"/>
    <XSL:apply-templates select="./E1PLOGI"/>
  </XSL:template>

  <XSL:template match="EDI_DC40">
```

```
  <Attribute name="IDOC_CTRL_DOCNUM">
    <XSL:for-each select="DOCNUM">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="IDOC_CTRL_MANDT">
    <XSL:for-each select="MANDT">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="IDOC_CTRL_DOCREL">
    <XSL:for-each select="DOCREL">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="IDOC_CTRL_IDOCTYP">
    <XSL:for-each select="IDOCTYP">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="IDOC_CTRL_SNDPOR">
    <XSL:for-each select="SNDPOR">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="IDOC_CTRL_RCVPOR">
    <XSL:for-each select="RCVPOR">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
</XSL:template>

<XSL:template match="E1PLOGI">
  <XSL:apply-templates select="./E1PITYP"/>
</XSL:template>

<XSL:template match="E1PITYP">
  <XSL:apply-templates select="./E1P0002"/>
  <XSL:for-each select="E1P0105">
    <Attribute name="PR_COMM_SUBTY">
      <XSL:for-each select="SUBTY">
        <Value>
           <XSL:value-of select="." />
        </Value>
      </XSL:for-each>
    </Attribute>
    <Attribute name="PR_COMM_USRID">
      <XSL:for-each select="USRID">
        <Value>
           <XSL:value-of select="." />
        </Value>
      </XSL:for-each>
    </Attribute>
    <Attribute name="PR_COMM_USRID_LONG">
      <XSL:for-each select="USRID_LONG">
```

```
        <Value>
           <XSL:value-of select="." />
        </Value>
      </XSL:for-each>
    </Attribute>
  </XSL:for-each>
</XSL:template>

<XSL:template match="E1P0002">
  <Attribute name="PR_PERNR">
    <XSL:for-each select="PERNR">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="PR_LASTNAME">
    <XSL:for-each select="NACHN">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="PR_FIRSTNAME">
    <XSL:for-each select="VORNA">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="PR_BIRTHDATE">
    <XSL:for-each select="GBDAT">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
</XSL:template>

</XSL:stylesheet>
```

If the IDOC message type was always going to be the USERCLONE message type, then you could use XSL like the following to get the required attribute mappings.

```
<XSL:stylesheet xmlns:XSL="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <XSL:output method="XML" indent="yes" />

  <XSL:template match="USERCLONE05">
    <DocRoot>
      <Entry>
        <XSL:apply-templates select="./IDOC"/>
      </Entry>
    </DocRoot>
  </XSL:template>

  <XSL:template match="IDOC">
    <XSL:apply-templates select="./EDI_DC40"/>
    <XSL:apply-templates select="./E1BPBNAME"/>
    <XSL:apply-templates select="./E1BPLOGOND"/>
    <XSL:apply-templates select="./E1BPADDR3"/>
    <XSL:apply-templates select="./E1BPLOGOND"/>
    <XSL:apply-templates select="./E1BPUSCOMP"/>
  </XSL:template>

  <XSL:template match="EDI_DC40">
    <Attribute name="TDI_DOCNUM">
      <XSL:for-each select="DOCNUM">
```

```
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="TDI_MANDT">
    <XSL:for-each select="MANDT">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="TDI_DOCREL">
    <XSL:for-each select="DOCREL">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="TDI_IDOCTYP">
    <XSL:for-each select="IDOCTYP">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="TDI_USERCLONE">
    <XSL:for-each select="USERCLONE">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="TDI_SNDPOR">
    <XSL:for-each select="SNDPOR">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
  <Attribute name="TDI_RCVPOR">
    <XSL:for-each select="RCVPOR">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
</XSL:template>

<XSL:template match="E1BPBNAME">
  <Attribute name="TDI_BAPIBNAME">
    <XSL:for-each select="BAPIBNAME">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
</XSL:template>

<XSL:template match="E1BPLOGOND">
  <Attribute name="TDI_CLASS">
    <XSL:for-each select="CLASS">
      <Value>
         <XSL:value-of select="." />
      </Value>
    </XSL:for-each>
  </Attribute>
```

```
    <Attribute name="TDI_TZONE">
      <XSL:for-each select="TZONE">
        <Value>
           <XSL:value-of select="." />
        </Value>
      </XSL:for-each>
    </Attribute>
  </XSL:template>

  <XSL:template match="E1BPADDR3">
    <Attribute name="TDI_FIRSTNAME">
      <XSL:for-each select="FIRSTNAME">
        <Value>
           <XSL:value-of select="." />
        </Value>
      </XSL:for-each>
    </Attribute>
    <Attribute name="TDI_LASTNAME">
      <XSL:for-each select="LASTNAME">
        <Value>
           <XSL:value-of select="." />
        </Value>
      </XSL:for-each>
    </Attribute>
    <XSL:apply-templates select="./E1BPADDR1"/>
  </XSL:template>

  <XSL:template match="E1BPADDR1">
    <Attribute name="TDI_E_MAIL">
      <XSL:for-each select="E_MAIL">
        <Value>
           <XSL:value-of select="." />
        </Value>
      </XSL:for-each>
    </Attribute>
  </XSL:template>

  <XSL:template match="E1BPUSCOMP">
    <Attribute name="TDI_COMPANY">
      <XSL:for-each select="COMPANY">
        <Value>
           <XSL:value-of select="." />
        </Value>
      </XSL:for-each>
    </Attribute>
  </XSL:template>

</XSL:stylesheet>
```

Configuration in SAP ALE Distribution Models

To enable the connector to be part of an SAP ALE distribution model the Connector must be setup as a logical system on the SAP system. To do this, two actions are required.

1. An RFC Destination of type TCP/IP is created where the Connector is registered as an external program. Take care to make sure that the program name provided in the RFC Destination, is the same value you give to the Connector configuration parameter "IDOC Server Program ID". To test the RFC connection ensure the Connector is running in an IBM Tivoli Directory Integrator AssemblyLine. This AssemblyLine may be bare bones one with only the connector, and possibly a script component to dump the resulting work entry attributes. Create RFC Destinations using SAP GUI transaction SM59.
2. A logical System is created that has the same name as the RFC Destination created in step 1. This is done using SAP GUI transaction SALE. You don't need to assign a client to the logical system like you do for other SAP logical systems that actually represent a real SAP System client.

Once the logical system is in place you can pretty much use it as you would any other logical system in an SAP ALE distribution model. The Connector has been tested as part of a SAP HR Master Data

distribution model, and as part of the pre-defined SAP CUA distribution model. If you run into other issues using other SAP ALE distribution models please contact IBM Support.

Troubleshooting the SAP ABAP Application Server Component Suite

Problems may be experienced for any of the following reasons:

SAP Java Connector not installed properly
Check the installation and re-install if necessary.

XSL Stylesheets not available
The Connectors rely on XSL stylesheets to perform their operations. See this note on problems that can occur if the XSL folder is not available in the Solution directory.

Missing `sapjco.jar`
If you attempt to use the SAP ABAP Application Server RFC FC and get an error similar to the following message:

```
13:01:58  Error in: InitConnectors: java.lang.ClassCastException:
     java.lang.NoClassDefFoundError

java.lang.ClassCastException: java.lang.NoClassDefFoundError
```

It may be that SAP JCo is not installed correctly. Check that `sapjco.jar` is in the *`Tivoli Directory Integrator_Home`*`/jars` directory. Refer to the instructions in "Configuring the SAP Java Connector" on page 446.

Missing `librfc32.dll`
If you attempt to use SAP ABAP Application Server FC and get an error similar to the following message:
"*The dynamic linked library LIBRFC32.dll could not be found in the specified path*"
On Windows machines, ensure that `librfc32.dll` is in the *`Tivoli Directory Integrator_Home`*`/libs` directory. On Solaris and AIX machines, ensure that `librfccm.{o/so}` has been added to the loadable library path.

Old version of `librfc32.dll`

If you get an error of the following type:

```
java.lang.ClassCastException: java.lang.ExceptionInInitializerError
```

It is possible that the `librfc32` being used is an older version and is not compatible with JCo 2.1.6. Check that there is no other `librfc32` in your PATH. Also check that any `librfc32*.{dll/so}` that is in your system path is at least version 6403.3.81.4751.

```
15:13:44  [YourAssemblyLine] BEGIN   selectEntries

15:13:45  [YourAssemblyLine] handleException: initialize,
java.lang.ClassCastException: java.lang.ExceptionInInitializerError

15:13:45  [YourAssemblyLine] initialize

java.lang.ClassCastException: java.lang.ExceptionInInitializerError
     at com.ibm.di.script.ScriptEngine.call(Unknown Source)
     at com.ibm.di.connector.ScriptConnector.selectEntries(Unknown Source)
     at com.ibm.di.server.AssemblyLineComponent.initialize(Unknown Source)
     at com.ibm.di.server.AssemblyLine.initConnectors(Unknown Source)
     at com.ibm.di.server.AssemblyLine.msInitConn(Unknown Source)
     at com.ibm.di.server.AssemblyLine.executeMainStep(Unknown Source)
     at com.ibm.di.server.AssemblyLine.executeMainLoop(Unknown Source)
     at com.ibm.di.server.AssemblyLine.executeAL(Unknown Source)
     at com.ibm.di.server.AssemblyLine.run(Unknown Source)
```

RFC_ERROR_SYSTEM_FAILURE: Screen output without connection to user
If the connector returns this message, please see SAP Note 49730 for more information.

Query Schema Issues
When performing a schema query using the Connectors with the IBM Tivoli Directory Integrator

GUI, an attempt to connect to the data source may result in an exception. These exceptions can be ignored. Any subsequent use of the **discover** schema button will succeed.

The Connectors do not support the *Get Next Entry* style of schema query. The Connectors support the torch button *Discover the Schema of the data source* style of schema discovery.

User Registry Company Code Assignment

If the value associated with the XML element, `<companyKeyName>`, does not represent a valid company code within SAP, or is not supplied at all, SAP will assign the configured default.

Changing Mode of Connectors Already in AssemblyLine

During testing, it was observed that changing the mode of Connector in the AssemblyLine did not always work. The Connector sometimes appeared to execute in its original mode, resulting in AssemblyLine errors. If this occurs, delete the Connector and add it to the AssemblyLine in the new mode.

Function Component differences to SE37 Test RFC Feature

In some cases, the RFC Function Component exhibits slightly different behavior to that observed when executing a given RFC from SAP's *Test Function Feature*, available from transaction SE37. In some cases, the SAP test feature will automatically convert values to internal German abbreviated values (for example, `BAPI_SALESORDER_GETLIST`). Therefore, some of the values returned by the connector in **Lookup** and **Iterator** mode may differ slightly from those returned by the SAP test function feature. When you are required to provide input XML files to set the values of parameters, you should supply the internal values (that is, the same format as the values returned by the connector in **Lookup** and **Iterator** modes).

The RFC Function Component will not pad out values of character string types to the maximum length.

User Registry Connector Warnings

In some cases, the Connectors may log warning severity messages as a result of application level ABAP warnings return from SAP. An example of warning messages logged by the User Registry Connector running in **Iterator** mode is shown below.

```
15:50:10  [newGetUsers] W: Unable to read the address (69) (D:\Program
   Files\IBM\IBMDirectoryIntegrator\xsl\bapi_user_get_detail_precall.xsl)

15:50:10  [newGetUsers] W: Unable to determine the company (76) (D:\Program
   Files\IBM\IBMDirectoryIntegrator\xsl\bapi_user_get_detail_precall.xsl)
```

In most cases, these warning messages can be ignored.

User Registry Connector In Update Mode

When run in this mode, the Connector expects the **sapUserName** attribute to be defined in the **Link Criteria** and as an XML element, `<sapUserName>`, within the value associated with the attribute **sapUserXml**. The values of **sapUserName** should match in both cases. The Connector does not verify the equality.

Password Behavior In SAP

After a new user is created in SAP, or the password of an existing user is changed, SAP will prompt that user to reset their password at the next logon. This is standard SAP behavior and occurs if the user is created or modified through the SAP transaction SU01, or the Connector.

Delete HR Personal Data With HR Connector

In some cases, an attempt to delete a Personal Data entry using the Connector, or SAP transaction PA30, may fail. The failure message states "*Record cannot be deleted (time constraint 1)*". Currently, there is no known solution to this problem.

Supplemental information for the SAP ABAP Application Server Component Suite

Example User Registry Connector XML Instance Document

The following code sample shows an example User Registry Connector XML Instance Document:

```
<User>
      <sapUserName></sapUserName>
      <sapUserPassword></sapUserPassword>
      <sapUserAlias>
          <aliasName></aliasName>
      </sapUserAlias>
      <sapAddress>
          <title></title>
          <academicTitle></academicTitle>
          <firstName></firstName>
          <lastName></lastName>
          <namePrefix></namePrefix>
          <nameFormat></nameFormat>
          <nameFormatRuleCountry></nameFormatRuleCountry>
          <isoLanguage></isoLanguage>
          <language></language>
          <searchSortTerm></searchSortTerm>
          <department></department>
          <function></function>
          <buildingNumber></buildingNumber>
          <buildingFloor></buildingFloor>
          <roomNumber></roomNumber>
          <name></name>
          <name2></name2>
          <name3></name3>
          <name4></name4>
          <city></city>
          <postCode></postCode>
          <poBoxPostCode></poBoxostCode>
          <poBox></poBox>
          <street></street>
          <streetNumber></streetNumber>
          <houseNumber></houseNumber>
          <country></country>
          <countryIso></countryIso>
          <region></region>
          <timeZone></timeZone>
          <primaryPhoneNumber></primaryPhoneNumber>
          <primaryPhoneExtension></primaryPhoneExtension>
          <primaryFaxNumber></primaryFaxNumber>
          <primaryFaxExtension></primaryFaxExtension>
      </sapAddress>
      <sapCompany>
          <companyNameKey></companyNameKey>
      </sapCompany>
      <sapDefaults>
          <startMenu></startMenu>
          <outputDevice></outputDevice>
          <printTimeAndDate></printTimeAndDate>
          <printDelete></printDelete>
          <dateFormat></dateFormat>
          <decimalFormat></decimalFormat>
          <logonLanguage></logonLanguage>
          <cattTestStatus></cattTestStatus>
          <costCenter></costCenter>
      </sapDefaults>
      <sapLogonData>
          <validFromDate></validFromDate>
          <validToDate></validToDate>
```

```
        <userType></userType>
        <userGroup></userGroup>
        <accountId></accountId>
        <timeZone></timeZone>
        <lastLogonTime></lastLogonTime>
        <codeVerEncryption></codeVerEncryption>
    </sapLogonData>
    <sapSncData>
        <printableName></printableName>
        <allowUnsecure></allowUnsecure>
    </sapSncData>
    <sapUserGroupList>
        <group>
            <name></name>
        </group>
        <group>
            <name></name>
        </group>
    </sapUserGroupList>
    <sapParameterList>
        <parameter>
            <parameterId></parameterId>
            <parameterValue></parameterValue>
        </parameter>
        <parameter>
            <parameterId></parameterId>
            <parameterValue></parameterValue>
        </parameter>
    </sapParameterList>
    <sapUserEmailAddressList>
        <email>
            <defaultNumber></defaultNumber>
            <smtpAddress></smtpAddress>
           <isHomeAddress></isHomeAddress>
            <sequenceNumber></sequenceNumber>
        </email>
        <email>
            <defaultNumber></defaultNumber>
            <smtpAddress></smtpAddress>
            <isHomeAddress></isHomeAddress>
            <sequenceNumber></sequenceNumber>
        </email>
    </sapUserEmailAddressList>
    <sapRoleList>
        <role>
            <name></name>
            <validFromDate></validFromDate>
            <validToDate></validToDate>
        </role>
        <role>
            <name></name>
            <validFromDate></validFromDate>
            <validToDate></validToDate>
        </role>
    </sapRoleList>
    <sapProfileList>
        <profile>
            <name></name>
        </profile>
       <profile>
            <name></name>
        </profile>
    </sapProfileList>
</User>
```

XSchema for User Registry Connector XML

The XSchema for User Registry Connector XML is show below:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="User">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="sapUserName" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="sapUserPassword" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="sapUserAlias" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="sapAddress" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="sapCompany" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="sapDefaults" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="sapLogonData" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="sapSncData" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="sapUserGroupList" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="sapParameterList" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="sapUserEmailAddressList" minOccurs="0"
                     maxOccurs="1"/>
                <xsd:element ref="sapRoleList" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="sapProfileList" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="academicTitle">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="20"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="accountId">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="12"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="aliasName">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="40"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="allowUnsecure">
     <xsd:simpleType >
      <xsd:restriction base="xsd:boolean">
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="buildingFloor">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="10"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="buildingNumber">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="10"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="cattTestStatus">
```

```
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="1"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="companyNameKey">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="42"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="costCenter">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="8"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="dateFormat">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="1"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="decimalFormat">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="1"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="defaultNumber">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="1"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="department">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="40"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="email">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="defaultNumber" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="smtpAddress" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="isHomeAddress" maxOccurs="1" minOccurs="0"/>
            <xsd:element ref="sequenceNumber" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="firstName">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="40"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="function">
 <xsd:simpleType >
```

```
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="40"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="group">
    <xsd:complexType>
        <xsd:sequence>
   <xsd:element name="name">
    <xsd:simpleType >
     <xsd:restriction base="xsd:string">
      <xsd:maxLength value="12"></xsd:maxLength>
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="isHomeAddress">
 <xsd:simpleType >
  <xsd:restriction base="xsd:boolean">
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="isoLanguage">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="2"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="language">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="1"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="lastLogonTime">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:minLength value="8"></xsd:minLength>
   <xsd:maxLength value="8"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="lastName">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="40"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="logonLanguage">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="1"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="name">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="40"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
```

```
<xsd:element name="name2">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="40"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="name3">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="40"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="name4">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="40"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="nameFormat">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="2"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="nameFormatRuleCountry">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="3"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="namePrefix">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="20"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="outputDevice">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="4"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="parameter">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="parameterId" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="parameterValue" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="parameterId">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="20"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="parameterValue">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
```

```
   <xsd:maxLength value="18"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="poBox">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="10"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="postCode">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="10"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="primaryFaxExtension">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="10"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="primaryFaxNumber">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="30"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="primaryPhoneExtension">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="10"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="primaryPhoneNumber">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="30"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="printDelete">
 <xsd:simpleType >
  <xsd:restriction base="xsd:boolean">
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="printTimeAndDate">
 <xsd:simpleType >
  <xsd:restriction base="xsd:boolean">
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="printableName">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="255"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="profile">
    <xsd:complexType>
```

```
        <xsd:sequence>
   <xsd:element name="name">
    <xsd:simpleType >
     <xsd:restriction base="xsd:string">
      <xsd:maxLength value="12"></xsd:maxLength>
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="region">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="3"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="role">
    <xsd:complexType>
        <xsd:sequence>
   <xsd:element name="name">
    <xsd:simpleType >
     <xsd:restriction base="xsd:string">
      <xsd:maxLength value="30"></xsd:maxLength>
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:element>
            <xsd:element ref="validFromDate" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="validToDate" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="roomNumber">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="10"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="sapAddress">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="title" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="academicTitle" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="firstName" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="lastName" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="namePrefix" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="nameFormat" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="nameFormatRuleCountry" minOccurs="0"
                 maxOccurs="1"/>
            <xsd:element ref="isoLanguage" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="language" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="searchSortTerm" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="department" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="function" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="buildingNumber" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="buildingFloor" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="roomNumber" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="name" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="name2" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="name3" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="name4" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="postCode" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="poBox" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="street" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="region" minOccurs="0" maxOccurs="1"/>
```

```
            <xsd:element ref="timeZone" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="primaryPhoneNumber" minOccurs="0"
                 maxOccurs="1"/>
            <xsd:element ref="primaryPhoneExtension" minOccurs="0"
                 maxOccurs="1"/>
            <xsd:element ref="primaryFaxNumber" minOccurs="0"
                 maxOccurs="1"/>
            <xsd:element ref="primaryFaxExtension" minOccurs="0"
                 maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="sapCompany">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="companyNameKey" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="sapDefaults">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="startMenu" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="outputDevice" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="printTimeAndDate" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="printDelete" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="dateFormat" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="decimalFormat" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="logonLanguage" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="cattTestStatus" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="costCenter" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="sapLogonData">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="validFromDate" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="validToDate" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="userType" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="userGroup" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="accountId" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="timeZone" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="lastLogonTime" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="sapParameterList">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" ref="parameter"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="sapProfileList">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" ref="profile"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="sapRoleList">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" ref="role"/>
        </xsd:sequence>
    </xsd:complexType>
```

```
</xsd:element>
<xsd:element name="sapSncData">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="printableName" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="allowUnsecure" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="sapUserAlias">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="aliasName" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="sapUserEmailAddressList">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" ref="email"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="sapUserGroupList">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" ref="group"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="sapUserName">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="12"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="sapUserPassword">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="8"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="searchSortTerm">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="20"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="sequenceNumber">
 <xsd:simpleType >
  <xsd:restriction base="xsd:nonNegativeInteger">
   <xsd:totalDigits value="3"></xsd:totalDigits>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="smtpAddress">
 <xsd:simpleType >
  <xsd:restriction base="xsd:string">
   <xsd:maxLength value="241"></xsd:maxLength>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
<xsd:element name="startMenu">
 <xsd:simpleType >
```

```
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="20"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="street">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="60"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="timeZone">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="6"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="title">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="30"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="userGroup">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="12"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="userType">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="1"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="validFromDate">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="10"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
    <xsd:element name="validToDate">
     <xsd:simpleType >
      <xsd:restriction base="xsd:string">
       <xsd:maxLength value="10"></xsd:maxLength>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>
</xsd:schema>
```

Chapter 6. Asset Integration Suite

IBM started their asset data integration initiative as a way to achieve integration in a loosely coupled multi-product environment. Its key goals are:

- Consistent data representation across products
- Improved data sharing through common mechanism
- Reduced data redundancy

The Tivoli CDM (Common Data Model), IT Registry (Data Integration Service) and IdML (Identity Markup Language) were born as a consequence of this initiative.

Note: The term "IT Registry" refers to an IBM product which permits centralized handling of IT resources. Please note that this term is not officially approved and can be changed in future.

Overview

The Common Data Model (CDM) is a consistent, integrated, logical data model that defines the general characteristics of information stored in the IT registry. The model specifies how this data is organized to correspond to real-world entities and defines the relationships between the entities. The CDM represents management information in a way that is easy for consuming management applications to use.

CDM components

Based on the Unified Modeling Language (UML), the CDM represents management information in terms of entities (called ManagedElements or Configuration Items) and the relationships among those entities. The CDM strives to include information from all of the logical models in use (such as CIM, BPEL, ITIL, SNA, and TMf) and integrates them into a single consistent model. The CDM and associated documents can be viewed at the Tivoli CDM Web site, which is included on the Tivoli Directory Integrator product DVD. To access the Web site, copy and unzip the `CDMWebsite.zip` file from the Tivoli Directory Integrator DVD. The CDM draws most of its concepts from UML, and the contents of the model can be used in UML development tools, such as IBM Rational® Software Architect.

Attributes

At the most basic level of granularity, the CDM represents atomic data as an attribute, as defined by UML. An attribute has an associated data type, a possible default value, and a specification of whether the attribute is single-valued or multi-valued. Certain data types, and enumerations, limit the actual values that an attribute can contain. All attributes in the CDM are globally defined, which means that an attribute with the same name has the same meaning, regardless of the context in which it is used. This is to foster consistent definition and use of the attribute in various environments and circumstances, such as events.

Following are examples of attributes:

- Manufacturer
- MemorySize
- PrimaryOwner
- PrimaryMacAddress

Classes

Attributes are grouped within the CDM into entities that correspond to items in the real world, such as computers, users, or business processes. This grouping of attributes is called a class. Classes in the CDM are arranged into a single-inheritance hierarchy that enables attributes to be shared among classes.

In some cases, classes are abstract: abstract classes contain common characteristics of entities, but instances of these classes cannot be created. `ModelObject`, the root of the class hierarchy, is an example of an abstract class. A vast majority of the classes in the CDM are concrete, which means that instances of them can be created in the IT registry.

Note that the class hierarchy of the CDM is rooted in the class `ModelObject`, not `ConfigurationItem`. In addition to CIs, other kinds of data will be stored in the IT registry and modeled using the CDM.

Interfaces

Many situations that commonly occur in the real world lead people to use multiple inheritance, which is supported by UML but not by the CDM. In order to handle these situations, the CDM includes the concept of an interface, which is a consistent collection of attributes (or a consistent source or target of a relationship) that can be "included" in a class definition anywhere in the class hierarchy. This is similar to the way in which Java handles interfaces, except that the CDM includes only data, not methods.

Interfaces themselves can be derived from other interfaces, thus forming another inheritance hierarchy. However, while an interface hierarchy can have multiple roots, the derivation hierarchy cannot mix interfaces and classes. Classes can be derived only from other classes, and interfaces can be derived only from other interfaces.

Relationships

One of the most important purposes of the IT registry is to store relationships between entities in the real world. The CDM therefore places a lot of focus on the definition of relationships between classes and interfaces, and assigns a specific semantic meaning to the relationship. For example, a relationship called "runsOn" may represent the fact that a piece of software executes in a particular environment.

Relationships in the CDM are related to, but differ from, a similar concept in UML called associations. An association is a semantic link between classes in UML; an example is a realization, where one entity makes a particular interface available. Nothing in UML forces a user to express the meaning of an association; you can simply draw a line between two entities. In the CDM, all associations (other than generalization and realization) are named or typed. The name of the association gives it a corresponding meaning, therefore making the association a relationship. All associations with the same name have the same meaning.

Naming and identification

In addition to representing and storing relationships between entities, the IT registry provides a correlation mechanism between entities. For example, two management products might discover a single computer system and call them different names; it is important to represent this as a single entity. In order to foster consistent identification of entities in the IT registry, the CDM formally defines the ways in which each type of entity (each class) is identified. To do so, the model uses *naming rules*.

Naming rules list the attributes that provide identifying characteristics, the combination of attributes are needed to identify the entity, and the context that makes the identification unique. Following are two examples of naming rules:

- Combining "Manufacturer", "MachineType", "Model", and "SerialNumber" gives a unique identification of a computer.
- The "DriveLetter" of a logical disk gives a unique identification of the disk within the context of an operating system.

Correlation in the IT registry is fostered by a consistent use of these rules and an understanding of which rules identify instances of the same type. When multiple names for the same instance arise, they are called aliases, and the IT registry represents the duplicates as a single instance. Consistent formation of names using the naming rules also allows the IT registry (or applications) to generate useful binary tokens known as globally unique identifiers (GUIDs) for the instances.

IT registry

To incorporate the CDM ideas of handling management information – its representation, naming and identification, Tivoli Directory Integrator relies on an IT registry. In essence, the IT registry implements these ideas and provides a way to work with the managed data. It consists of two parts:

- A centralized database, which contains the registered resources and the CDM meta-data (definitions of the CDM classes, relationships, naming rules, and so forth). By default TDI expects that such a database is set up and available for remote use, but it also provides everything needed for setting up a local instance. See section “IT Registry database setup” on page 517 for details.
- A set of services that provide convenient Java API for performing the different activities of the data integration process. Tivoli Directory Integrator exploits these Java APIs directly in some of its Connectors, Function Components, and Parsers

The most important functionality provided by the IT registry is naming and reconciliation. Let us assume that we have two products that manage the same resource, but identify it differently based on their capabilities. Then the two products cannot effectively work together in collaborative fashion. Each of them will have only a subset of the resource's data, so they will not know it is the same resource at all. To solve this situation, users can rely on the combination of Tivoli Directory Integrator and IT registry. Tivoli Directory Integrator will communicate with each product, take that data and register it the IT registry using its Java API. This way the IT registry will handle the resource naming, representation and storage. Additionally, it will employ the available Naming Rules and check if the two products are "talking" about the same resource. This way, only a single resource will be kept in IT registry and it will contain the information obtained from both products.

Another possible issue in this scenario is that the two products can use different terms to describe the same entity (for example, use both "IBM" and "IBM Corporation" to signify the resource's manufacturer). This is handled by the IT registry through a simple string-mapping functionality. For the key term "IBM" there is a set of acceptable representations (for example, "IBM", "IBM Corporation", "IBM Corp") and if the provided values match any of them the key term is returned. Thus, the original value has been cleansed. The result of using this solution is a clearer, smaller, and more consistent resource representation that can be used by multiple other products.

Besides the naming and reconciliation, TDI relies on IT registry for providing the Common Data Model meta-data. When registering resources, users need to specify their class and attributes. Without knowing the ones supported (or required) by the CDM this would be impossible. Thus, Tivoli Directory Integrator uses the IT registry's meta-data functionality to obtain the needed definitions and significantly ease users in this process.

Tivoli Directory Integrator provides Components that will utilize the described functionalities and permit their use in its integration solutions; see section “Components of the suite” on page 500 for details.

The IT registry provides a suitable way for handling resources in a unified manner. However, there is another technique for communication among software components that use the Common Data Model - Discovery Library Adapters. These are runtime components that exploit mechanisms native to Management Software Systems (or OMPs) to extract specific details about resources and resource relationships. Then, they transform this information into files that conform to the IdML schema. The purpose of Discovery Library Adapters, therefore, is to discover and keep current sets of resources and relationships that comprise business applications and support business and infrastructure processes.

The IdML (Identity Markup Language) is the Discovery Library XML schema specification that provides a standard way for storing Management Software Systems (MSS) data and operation sets that define groups of operations for creating, updating, and deleting managed resources. For more details on IdML refer to section “Introduction” on page 500.

Components of the suite

Open IdML Function Component

Introduction

This is the first Component of the IdML suite – a set of Tivoli Directory Integrator Function Components, a Connector and a Parser designed to create and read IdML files (or books). The particular use of this Function Component is summed up in two points:

- Create an IdML book and statically share it, so that the other Components of the suite can access it. Alternatively, it can just get an existing book, provided that it is not currently in use by another Open IdML FC. For this purpose the Component uses the *Book Name* - an identifier to which it maps the actual IdML book, so that the other Components can look it up (assuming they know its name). To help beginner users, if no value is set in the configuration panel of the FC (or in its Output Map), a default book name (an empty string) is used.

  Since only the Open IdML FC can share books, it is also the one to free them from the static map, when they are no longer needed. In order to prevent data corruption (if two Open IdML FCs attempt to work with the same book) this Component places an exclusive lock on the associated book, thus preventing other Open IdML FCs to use it, until it is freed.
- Open the acquired IdML book. Once the Component has received access to the book, it attempts to open it – set some configuration parameters used by the other Components and write the heading of the IdML document. Since IdML is an XML derivate, it follows a unified schema, as can be seen below:

```
<?xml version="1.0" encoding="UTF-8"?>
<idml:idml xmlns:idml="http://www.ibm.com/xmlns/swg/idml"
  xmlns:cdm="http://www.ibm.com/xmlns/swg/cdm"
  xmlns:xsi"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/xmlns/swg/idml idml.xsd">
  <idml:source.IdMLSchemaVersion="0.8">
   <cdm:process.ManagementSoftwareSystem id="id_value" CDMSchemaVersion="2.9.3" >
    <cdm:MSSName>ibm-cdm:///CDMMSS/identifier</cdm:MSSName>
    <cdm:Label>label_value</cdm:Label>
    <cdm:ProductName>product_value</cdm:ProductName>
    <cdm:ManufacturerName>manufacturer_value</cdm:ManufacturerName>
   </cdm:process.ManagementSoftwareSystem>
  </idml:source>
  <idml:operationSet opid="1">
   OPERATIONS
   ...
   ...
  </idml:operationSet>
</idml:idml>
```

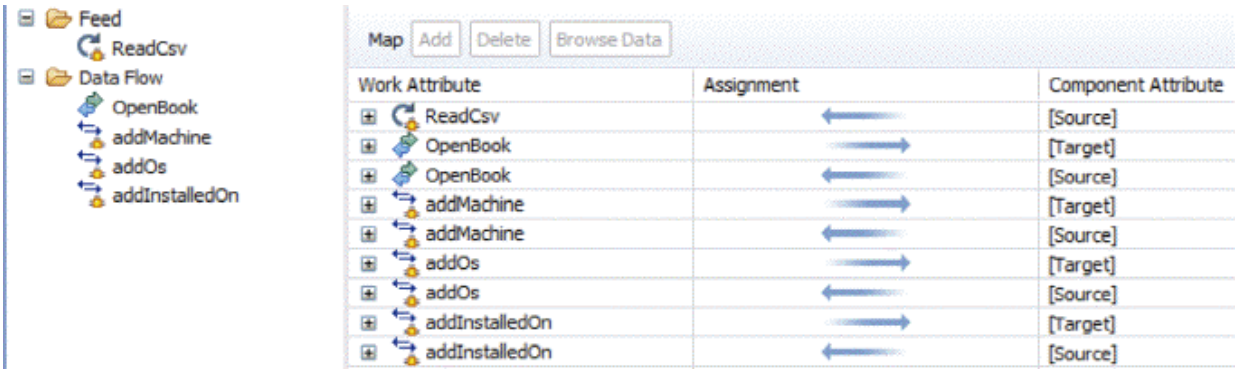
  Each IdML book has two parts – one `idml:source` (also referred as heading) and one or more `idml:operationSet`-s.

  The source part is designed to provide identifying information for the MSS (ManagementSoftwareSystem) that manages the resources listed in the IdML (in the operationSet part), while the operationSet(s) determine what should be done with these resources. The Open IdML FC is responsible for adding the MSS data (name, label, manufacturer, and so forth) to the IdML, when the book is opened. If the Open IdML FC has acquired a book that is already opened, it will not attempt to add the MSS source information, but instead log a message to the user and pass the execution to the next Component in the AssemblyLine.

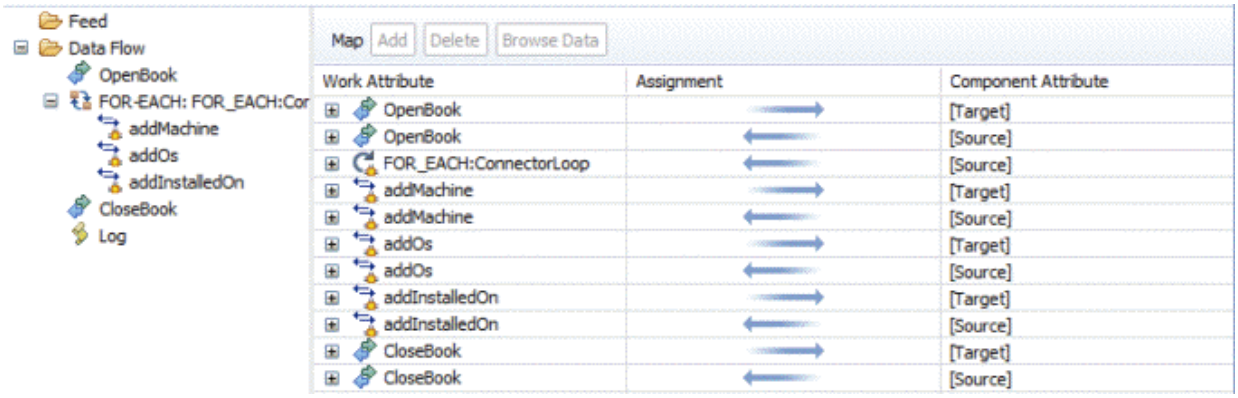
An additional function of this FC is to guarantee that the opened IdML book will be closed (the ending tag, `</idml:idml>` is added). By default this is the function of the "Close IdML Function Component" on page 504, but can be performed by the Open IdML FC, in case of a normal Feed-Flow AssemblyLine.

In fact, there are two AssemblyLine types in which the IdML suite can participate:

1. A normal Feed-Flow AssemblyLine, for example, the data is read from a Connector in Iterator mode in the Feed section of the AssemblyLine and passed to the Flow for manipulation (see below). Since all Components in the Flow section are executed repeatedly, this does not permit to use the Close IdML FC – it will attempt to close the book on each pass. Thus, its task is performed by the Open IdML FC, since it can be executed repeatedly. The result is that it opens the IdML book, during the first AL iteration, subsequently only logs information messages to the user and closes the book when the AL terminates.



2. A loop driven AssemblyLine – a Loop Component, fed by a Connector, repeatedly iterates over its enclosed Components passing them the different data read from the Connector. In this case the AssemblyLine performs only one iteration, while the Components inside the loop get executed numerous times. Here the IdML book closure is performed by the Close IdML Function Component, which can provide you with additional information for the generated book (for example, full path to the IdML file or its content).



The Open IdML FC supports two storage types for IdMLs:

- Standard IdML books stored as files. This approach is more memory efficient since the IdML is not accumulated in memory, but is directly flushed to the file.
- IdML snippets – valid IdML documents stored in memory. This way we skip the IdML file and return its contents directly. Of course this leads to higher memory consumption than storing in a file.

Depending of the Open IdML FC's configuration, either one can be generated (in the case of file stored ones, the user needs to specify where they will be stored or the current Solution Directory will be used).

As advanced configuration options, the FC accepts a custom Book Name, so that the default book will not be used (as mentioned before). This is particularly handy if the default book is already in use. Then you can simply specify another name for the IdML Components you want to use, and the AssemblyLine will run successfully. Moreover, the name of the used book can be overridden in the Output Map of the FC, thus allowing new names to be provided at run time. This can be very useful, if the Flow section of the AL (shown above) is responding to requests from different users (through some server Connector, for

instance). In this case, if several users request an IdML using the default book, only the first will succeed, while the others will have to wait for him to finish or get an error message. To solve this, the user can be asked to provide a unique identifier in its request, which to be used as a book name by the IdML suite.

Another advanced option is to generate a Refresh IdML, as opposed to a regular Delta one. In the IdML terminology there are two types of IdMLs:

- Delta ones, meaning that the operations listed in the IdML only modify the data already existing in the Configuration Management Database or CMDB (for example, Tivoli Application Dependency Discovery Manager), upon the importing of the book. Thus the document can specify resources to be added to the database (the CREATE operation), resources for which data is to be updated (the MODIFY operation) and resources to be removed (the DELETE operation).
- Refresh IdMLs. As opposed to the above one, this type can contain only CREATE operations meaning that it can only add resources to the CMDB. The difference is that the resources already present in the database will be cleared, leaving only the ones from the IdML.

The FC can also use the DL Certification Tool to validate the generated IdML, either as a file or in-memory.

Another detail of this Function Component is how it handles the Common Data Model (CDM). Since it adds information for a MSS in the IdML, and MSSs are in essence ordinary Configuration Items, the user working with the Component needs to know the names of the needed MSS attributes (for example, `cdm:MSSName`, `cdm:Hostname`). This information is provided by the CDM. In fact, two separate sources of CDM meta-data are supported by this Component:

- A local copy in the form of a JAR file shipped with Tivoli Directory Integrator (`idml_cert.jar`). There the CDM class names and attributes are stored as Java classes, which can be interpreted by the IdML Components.
- A remote IT Registry system, to which the FC can connect to and retrieve the needed attribute names, If this approach is chosen, the user can either rely on the common IT Registry credentials specified in the `etc/it_registry.properties` file, which are accessible to all ALs, or set custom ones specifically for that solution. See “The it_registry.properties file” on page 516 for more information on the IT registry properties.

An important limitation is that all attributes are displayed, without signifying which are needed by any of the naming rules for the `ManagementSoftwareSystem` class (also known as identifying attributes). To ease usage, the Open IdML FC expects the attribute `cdm:MSSName` to be mapped, or if it is not present the combination of `cdm:Hostname`, `cdm:Manufacturer` and `cdm:ProductName` (these are the identifying attributes required by the two naming rules of the `process.ManagementSoftwareSystem` class). If neither of them is provided, an exception will be thrown.

When dealing with the CDM attributes needed by the Open IdML FC, there are several important points. The *CDM Version* parameter specifies the CDM meta-data used when generating the IdML. Thus, it is best to rely on the provided button for setting its value. The available CDM attributes for the `process.ManagementSoftwareSystem` will be listed when the Query Schema functionality is used. Most of them specify properties of the MSS, but several affect other aspects as well. For instance, the `$id` attribute will override the default MSS ID formed by concatenating the Application Code and Hostname attributes and will change the name of the generated IdML file.

Also, if the `cdm:MSSName` attribute is not explicitly provided, the `cdm:Hstname`, `cdm:ManufacturerName` and `cdm:ProductName` are used to set the unique name of the MSS. If any of them is not provided either, an Exception will be thrown by the Component.

Finally, the `cdm:SourceToken` attribute can be used to specify the source token of the created MSS. Through it the you can specify how the MSS can be contacted (for example it can contain an HTTP URL that can be used to connect to the MSS and query for additional information).

Schema

Output Schema:

$idmlBookName
This attribute can be used to override the name of the book used by this Component.

applicationCode
This attribute can be used to override the value of the **Application Code** parameter from this FC's configuration panel. Its value should be specified either here, or by the applicationCode panel parameter; if neither are provided the name of the generated IdML file (if stored as a file) will start with an empty string instead of an application code.

CDM attributes of the MSS
Their names are retrieved from the used CDM. In order to create a valid IdML document, you must map either the `cdm:MSSName` attribute, or `cdm:Hostname`, `cdm:ProductName`, `cdm:ManufacturerName`.

Configuration

The Open IdML Function Component uses the following parameters:

Store IdML
Drop-down list; determines the storage mechanism for the book opened by this FC.

Directory Name
The name of the directory where the IdML book will be stored. Only relevant if the book will be stored as a file. If the field is left blank the Solution Directory will be used.

Application Code
The Application Code of the MSS registered at the beginning of the IdML. It can also be provided in the Output Map of the Component (in which case this value will be overridden).

Hostname
The Hostname of the MSS registered at the beginning of the IdML. It can also be provided in the Output Map of the Component (in which case this value will be overridden).

CDM version
The Common Data Model version used by the IdML. Its format is <version>.<release>.<modifier>. It can be discovered using the script button next to the field.

Book Name
The name of the IdML book that this Component will open. If left blank, the default IdML book will be opened.

Refresh
Determines whether this will be a refresh or delta IdML. The default is unchecked, that is, *false*.

Validate
Enables the usage of a validation tool for the generated IdML. The default is unchecked, that is, *false*.

Use IT Registry for CDM
Determines whether the FC will rely on IT Registry for the CDM meta data. The default is checked, that is, *true*.

JDBC URL
The JDBC URL used for connecting to the IT Registry database. Once you have provided all JDBC parameters, you can use the **Test Connection** button to test the JDBC connection to the IT Registry database.

JDBC Driver
The database driver used for connecting to the IT Registry database.

Username
The username used when connecting to the IT Registry database.

Password
The password used when connecting to the IT Registry database.

See also
The OPEN IdML Function component exposes a package for handling CDM meta-data – `com.ibm.di.fc.idml.md`. See the Javadocs for details.

Close IdML Function Component

Introduction
This Component attempts to close an already opened IdML book. If the book has been already closed, it just logs a message to the user. If the closing procedure was executed successfully, the Close IdML Component can provide additional information of the book in the `$idmlBook` attribute, and the book is no longer statically shared. It will contain either the full path to the IdML document (in case it is stored as a file), or its actual contents (for the in-memory IdML snippets).

As all Components from the IdML suite, this FC accepts a Book Name parameter, and will look up a different book, depending on its value. The name of the used book can also be overridden at runtime by the `$idmlBookName` attribute.

The Close IdML Function Component cannot be used in normal Feed-Flow AssemblyLines, since it will attempt to close the IdML book on each iteration. For more information see section "Open IdML Function Component" on page 500.

Schema

Output Schema:

$idmlBookName
This attribute can be used to override the name of the book used by this Component.

Input Schema:

$idmlBook
Depending on the type of generated IdML, this attribute will contain either the full path to the IdML file (for IdMLs stored as files), or the full content of the IdML (for in-memory IdML snippets).

Configuration
The Close IdML Function Component uses the following parameter:

Book Name
The name of the IdML book that this Component will close. If left blank, the default IdML book will be closed.

Rolling IdML Function Component

Introduction
The Rolling IdML Function Component is used when there is a need to limit the size of the generated IdML files. Thus it is only applicable for IdML documents stored as files. For in-memory IdML books, its usage will lead to an Exception.

The FC will attempt to close the IdML document when any of its conditions is met and open a new one where the next data to be stored. It performs these operations in the context of one book, so to the user it appears that one book is repeatedly rolled, not that several books are opened and closed. Each of the

resulting IdML files is a valid IdML document and contains the MSS data heading. Since the IdML file names are generated using the current time, this avoids the need to specify a name pattern for the rolled files.

When the FC has rolled the IdML book, it returns the name of the closed file as an attribute in its Input Map - `$idmlFileName`. Otherwise, the value of this attribute is null. If the IdML book was configured with the *Validate* option by the Open IdML FC, each rolled file will be validated upon its closure.

The Rolling IdML FC will attempt to perform its function based on two conditions:

- artifact count – the number of Configuration Items (CIs) or Relationships added to the IdML document. If this count is met (or exceeded) rolling will be performed.

  Note: the MSS itself is not considered a CI by this FC.
- file size – the actual size of the IdML file has reached a given size in kilobytes (KB).

  Note: Since for the closing of the file several tags must be added, be aware that the size of the resulting file can exceed this value with several bytes.

The default value for the above conditions is 0, meaning unlimited artifact count and file size. If you add this Component to an AssemblyLine, but do not specify specific values for the rolling conditions, the FC will not attempt to split the generated IdML file.

As with all Components from the IdML suite, this FC accepts a Book Name parameter, and will look up a different book depending on its value. The name of the used book can also be overridden at runtime by the `$idmlBookName` attribute.

Schema

Output Schema:

$idmlBookName
This attribute can be used to override the name of the book used by this Component.

Input Schema:

$idmlFileName
If this FC has performed its function, this attribute will contain the full path to the already closed IdML file. Otherwise, if no rolling has occurred, this attribute will have a *null* value.

Configuration

The Rolling IdML Function Component uses the following parameters:

Artifact Count
Determines the maximum number of artifacts to be stored in one IdML file (0 is considered unlimited).

File Size (KB)
Determines the maximum number of kilobytes to be written to one IdML file (0 is considered unlimited).

Book Name
The name of the IdML book that this Component will roll. If left blank, the default IdML book will be used.

IdML CI and Relationship Connector

Introduction

This Connector adds an artifact - either a Configuration Item (CIs) or a Relationship, to the IdML book. Thus, in its configuration the user needs to specify the desired artifact type and the class of the added item. In order to ease this task the user can directly discover the class names supported by the used CDM.

As with the Open IdML FC, the CDM meta-data can be retrieved from a local JAR file or by connecting to a remote IT Registry system. You can choose between these options in the Advanced section of the Connector's configuration panel. Other usability features are the ability to check the version of the used CDM, and test the connection to the remote IT Registry system (if it is being used). For the IT Registry case, you can also provide information how to connect to it (for example, JDBC URL, Driver, username and password). If these fields are left blank the default values specified in the `etc/it_registry.properties` file will be used.

The selected CDM source also affects the attributes displayed when querying the schema of the Output Map of the Connector. For instance, if the IT Registry CDM is used, when listing a CI's attributes you will get not only the specific attributes of its class (as is when the JAR meta-data is used), but also those of its parent classes. This leads to a somewhat slower response than when the JAR definitions are used. This feature is most notable when listing the Relationship types in the Connector's configuration panel. If the IT Registry CDM is used, you are able to see additional information, specifying which classes of CIs can act as sources and which as targets of the chosen Relationship. This can be very useful if you are unaware of the restrictions of the needed Relationship, but is a fairly slow operation which requires much more time than if the JAR CDM is used (then only the Relationship types will be listed, without the class restrictions).

The naming rules limitation is also valid for this Connector. You are able to see all attributes of the chosen CI class but can not determine which ones are part of a naming rule (also known as identifying attributes) and what rule exactly. Thus, in order to satisfy the required attributes of a CI, must find information for them in the CDM.

As with all Components of the IdML suite, this Connector accepts a Book Name parameter, and will look up a different book, depending on its value. The name of the used book can also be overridden at runtime by the `$idmlBookName` attribute.

Another important parameter which can be provided in the Connector's Output Map is the `$operation` attribute. It determines what operation will be performed with the specified CI/Relationship, when the IdML file is imported into a CMDB. It can be either added to the CMDB (the CREATE operation), updated (MODIFY), or removed (DELETE). These values – CREATE, MODIFY and DELETE (case insensitive) can be set to the `$operation` attribute. Note that if the used IdML is opened as a Refresh one (see section "Open IdML Function Component" on page 500) only the CREATE operation is supported and passing another value will cause an Exception. If you specify no value for the `$operation` attribute, the CREATE value will be used by default.

The other option for setting the IdML operation is to pass a delta enabled work entry to the Connector. Since the IdML Ci and Relationship Connector is "delta aware", it will interpret the delta operation set to the entry and map its value to the IdML operations. The mapping is fairly straightforward:

Table 73. Delta codes to IdML Operations mapping

Delta operation	IdML operation
ADD (`Entry.OP_ADD`)	CREATE
MODIFY (`Entry.OP_MOD`)	MODIFY
DELETE (`Entry.OP_DEL`)	DELETE

Keep in mind that the provided delta operation will ALWAYS override the value of the `$operation` attribute.

When you query the schema of the Output Map of this Connector, the attributes of the chosen CI/Relationship class will be listed. For the CI case they will include the `$id` and `cdm:SourceToken` attributes. The `$id` permits you to override the default value of the CI's ID, and supply a custom one (any string can be used as an ID). The default ID is an integer identifier that is incremented each time an artifact is added to the IdML book. This ensures the ID's uniqueness, while if you provide a custom one, you must guarantee it will not overlap an existing one. If the IdML book contains several CIs with the same ID, this can cause problems when importing it to a CMDB. Since the ID is used to determine the CIs participating in a Relationship, having several of them with the same identifier will produce corrupt results. Such a problem can be detected earlier by enabling the book certification feature of the Open IdML FC.

The `$id` attribute will be returned by the Input Map of the Connector as well, so that you can map it directly to another IdML Ci and Relationship Connector, configured to add a Relationship.

The `cdm:SourceToken` attribute can be used to provide a unique identifier for the added CI. While the `$id` attribute is unique in the IdML book, the `cdm:SourceToken` must be unique in the whole realm of the MSS. It can later be used to uniquely identify and locate a CI by the IT Registry.

Querying the schema of a Relationship using the IdML Ci and Relationship Connector will return only two attributes:

- `source` - this is the id of the CI that is the source of the Relationship
- `target` - this is the id of the CI that is the target of the Relationship

Both of these attributes must be provided when defining a Relationship, otherwise an exception will be thrown.

There is one additional type of attributes supported by the IdML Ci and Relationship Connector: *extension attributes*. As opposed to regular ones, these are stored in a sub-element of the artifact's element in the IdML XML document and are used to provide additional information for each Configuration Item. Since they do not need to comply with the CDM schema of that CI, users can rely on them to map any specific data. Such attributes are distinguished by their names, which follow this template: `cdm:extattr:AttributeName` (for example, `cdm:extattr:Testing`).

The IdML Ci and Relationship Connector provides additional functionality when dealing with IdML snippets. This can be especially useful when an AssemblyLine is configured to send the generated IdML using an HTTP Server Connector. In the normal case, the whole IdML should be accumulated, before it can be mapped to the `$idmlBook` attribute of the Close IdML FC and returned to the caller. However, the IdML Ci and Relationship Connector provides a method: `resetBook()`, that can be used to retrieve the current content of the IdML book and send it using the HTTP Server Connector. This way, the response will be chunked and the caller will not have to wait for the whole IdML to be completed, but will get the data as soon as it is ready. If this method is called for an IdML book stored as a file, *null* is returned.

Schema

Output Schema:

$idmlBookName
: This attribute can be used to override the name of the book used by this Connector.

$operation
: This attribute determines the IdML operation that is to be performed with the specified CI/Relationship. If no value is specified, "create" will be used. Also, if a delta tagged entry is passed to the Connector, its delta operation will override the one specified by the `$operation` attribute.

CDM attributes of the CI/Relationship
Their names are retrieved from the used CDM, in order to create a valid IdML document.

$id This attribute is present only when querying the schema of a CI. It contains an identifier unique in the IdML document.

cdm:SourceToken
This attribute is present only when querying the schema of a CI. It contains an identifier unique in the whole realm of the MSS managing the resource.

source This attribute is present only when querying the schema of a Relationship, this is the id of the CI that is the source of the Relationship.

target This attribute is present only when querying the schema of a Relationship, this is the id of the CI that is the target of the Relationship.

Input Schema:

$id This attribute contains the id given to the created Configuration Item. Its value can be either from the internal counter provided by the book or the one specified in the Output Map of the Connector. If the Connector is creating a Relationship, this attribute will have a *null* value.

Configuration
The IdML Ci and Relationship Connector uses the following parameters:

Artifact Type
Drop-down list; determines the type of artifact that this Connector will add to the IdML file.

Class Type
The type of Configuration Item or Relationship that will be created. You can use the **Select...** button to enter one of the pre-defined types.

Book Name
The name of the IdML book that will be used by this Connector. If left blank, the default IdML book will be used.

Use IT Registry for CDM
Determines whether the FC will rely on IT Registry for the CDM meta data. You can use the Test Connection to verify that you can reach the IT Registry system.

JDBC URL
The JDBC URL used for connecting to the IT Registry database. Once you have provided all JDBC parameters, you can use the **Test Connection** button to test the JDBC connection to the IT Registry database.

JDBC Driver
The database driver used for connecting to the IT Registry database.

Username
The username used when connecting to the IT Registry database.

Password
The password used when connecting to the IT Registry database.

IdML Parser

Introduction
The purpose of the IdML Parser is to parse the contents of an IdML file. It can only be used for reading IdML documents, while the Open IdML FC, IdML Ci and Relationship Connector, Close IdMl FC and Rolling IdML FC should be used for creating them. It relies on the XML Parser for handling the IdML files and snippets.

Please see section "Open IdML Function Component" on page 500 to see more information about the schema of the IdML XML.

The MSS section of an IdML XML is parsed only during the first iteration of the Component and the received MSS data is returned on every subsequent iteration. With each iteration the Parser also reads a single artifact (either a CI or a Relationship) from the operationSet section, parse its attributes and map them to the returned entry. Along with regular CDM attributes like `cdm:Manufacturer`, `cdm:SourceToken`, and so forth, the Parser reads the artifact's extension attributes and returns them as well. The only difference is that instead of `cdm:AttributeName`, they will be named `cdm:extattr:ExtendedAttributeName`.

The input file/snippet passed to the IdML Parser can be either a Delta or a Refresh IdML. For more information about IdML types refer to section "Open IdML Function Component" on page 500. Its type will determine the value of the `$idmlType` attribute in the Input Map of the Parser. The supported values are DELTA and REFRESH (case insensitive).

Depending on the type of artifact read by the Parser, the `$artifactType` attribute can be either CI (for Configuration Items) or Relationship. Both values are case insensitive.

Similarly, the class type will be mapped to the `$classType` attribute in the Input Map. It will have values like `cdm:ComputerSystem`, `cdm:OperatingSystem`, and so forth for CIs, and `cdm:installedOn`, `cdm:runsOn` and so forth for Relationships.

The value of the `$operation` attribute of the input map can be CREATE, MODIFY or DELETE (case insensitive). It is determined based on the operation element of the IdML document where the artifact was read from.

Schema

Output Schema:

This parser does not have any Output Schema.

Input Schema:

$operation
This attribute is used for determining the value of operation in the IdML XML schema.

$idmlType
This attribute determines whether a given IdML XML is a normal (delta) or refresh IdML file.

$classType
This attribute determines the class type of a CI or Relationship artifact.

$artifactType
This attribute decides whether it is a CI or a Relationship.

$cdmVersion
This attribute contains the value of CDM version IdML XML is using.

$id This attribute contains the value of the *id* attribute of a CI.

mss.*attributeName*
All MSS attributes will be mapped and their names will be prefixed with the token `mss`.

cdm:SourceToken
This attribute contains the value of the *sourceToken* attribute of a CI.

source This attribute contains the value of the *source* attribute of a Relationship.

target This attribute contains the value of the *target* attribute of a Relationship.

Configuration

The Parser has the following parameters:

Character Encoding
Character Encoding to be used.

Detailed log
Check this parameter to enable additional log messages.

Comment
This parameter can hold any user comments. It is not taken into account during the Function Component operation.

Data Cleanser Function Component

Introduction

The Data Cleanser Function Component relies on the IT Registry to clean the value of a string provided as its `$inputString` attribute. However, before attempting this operation, you need to specify the type of CDM attribute that this value belongs to (for example, `cdm:Manufacturer`, `cdm:ProductName`, and so forth). By pressing the **Select...** button in the configuration panel of the FC, you can easily see a list of all the CDM attribute types supported by the IT Registry.

If no cleansing rule exists for the provided string, it will be returned unmodified in the Input Map of the FC.

Schema

Output Schema:

$cdmAttributeType
This attribute can be used to override the value of the attribute type, provided in the configuration panel of the FC.

$inputString
This attribute will contain the String that needs to be cleansed.

Input Schema:

$cleansedString
This attribute contains the cleaned value of the `$inputString` attribute given in Output Map of the FC. If no cleansing rule can be found for it, the original string is returned.

Configuration

The Data Cleanser Function Component's Configuration panel uses the following parameter:

Attribute Type
The Common Data Model Attribute Type of the String which needs to be cleansed. You can use the **Select...** button to enter the value of one of the pre-defined attribute types.

Init IT Registry Function Component

Introduction

The Init IT registry Function Component registers a Management Software System (MSS) to the IT registry database and returns the GUID of registered MSS as an Input Map attribute. However, the GUID is wrapped as a `com.ibm.di.fc.itregistry.ConfigurationItemId` object. The reason is that you must not work directly with GUIDs or view their contents. Instead, you should simply map the `ConfigurationItemId`-s to other Components from the IT registry suite, when this is needed (for example, when registering a Relationship).

This Component will bypass IdML files, commonly used for transferring data between CMDBs. Instead of creating such a file, which then must be bulk loaded in the IT Registry, this Component can directly import its data to the IT Registry database. As its equivalent in the IdML suite, the Init IT registry FC will rely on static sharing of a book (in this case meaning, sessions of operations performed on the IT Registry DB). By using it, the IT Registry Components will share information about the performed operations. It includes a flag denoting whether a Refresh operation should be performed by the Components using the book (meaning that only artifacts registered by them should be kept in the database and all older ones should be removed) and a timestamp marking the moment when the Init IT registry FC was executed (used for separating the old from the new artifacts in the database, during a Refresh). To determine which book should be used, you can either provide a value for the Book Name field in the configuration panel of the FC or through the `$itRegistryBookName` attribute in its Output Map.

Since this Component connects to the IT Registry database, it relies on the values specified in the `etc/it_registry.properties` file (JDBC URL, JDBC Driver, username and password). To allow easier modifications of these properties each Component has fields for overwriting their values in its Advanced configuration section. For more information on the IT Registry properties, see "The it_registry.properties file" on page 516.

The flag **Use IT registry for CDM** in the configuration panel of the FC is used to indicate whether to use the local JAR or an IT Registry system for retrieving CDM's meta-data definitions.

By pressing the Connect button in the Output Map of the FC, you can populate its Output Schema. This way, all the CDM attributes supported by the `process.ManagementSoftwareSystem` class will be displayed and you can directly map them (even without exactly knowing their names).

The naming rules limitation is valid for this Function Component. You will be able to see all attributes of the `ManagementSoftwareSystem` but can not determine which ones are part of a naming rule and what rule exactly. To ease usage, the Init IT registry FC will expect the attribute `cdm:MSSName` to be mapped, or if it is not present the combination of `cdm:Hostname`, `cdm:Manufacturer` and `cdm:ProductName` (these are the identifying attributes required by the two naming rules of the `process.ManagementSoftwareSystem` class). If neither of them is provided, an exception will be thrown.

Schema

Output Schema:

$itRegistryBookName
This attribute can be used to override the name of the book used by this Component.

CDM attributes of the MSS
Their names are retrieved from the used CDM. In order to create a valid IdML document, the you must map the `cdm:MSSName` attribute or the `cdm:Hostname`, `cdm:ProductName`, `cdm:ManufacturerName` attributes.

Input Schema:

$mssGuid
This attribute is a `com.ibm.di.fc.itregistry.ConfigurationItemId` object that contains the GUID of the registered MSS.

Configuration

The Init IT Registry Function Component uses the following parameters:

CDM version
This parameter specifies the Common Data Model Version in the form of `version.release.modifier` (three dot-separated integers). The version should match with the IT Registry DB CDM version. As long as the CDM version is the same, different CDM release or modifier are backwards compatible.

Detailed log
Check this parameter to enable additional log messages.

Comment
This parameter can hold any user comments. It is not taken into account during the Function Component operation.

BookName
The BookName parameter is shared statically between the IT Registry Components, and each of them will perform its modifications in its context. Each will specify a book name in its configuration (or leave the field blank you want to use the default book) and will retrieve the corresponding MSS GUID. This attribute must be specified either in the configuration panel or in the output map of the FC.

Refresh
If selected (that is, *true*) it causes the book (Book Name) to perform a refresh of the IT Registry database. The default is *false*.

Use IT Registry for CDM
Determines whether the FC will rely on IT Registry for the CDM meta data.

JDBC URL
The JDBC URL used for connecting to the IT Registry database.

JDBC Driver
The database driver used for connecting to the IT Registry database.

Username
The username used when connecting to the IT Registry database.

Password
The password used when connecting to the IT Registry database.

All JDBC-related parameters derive their default values from the `etc/it_registry.properties` file.

IT Registry CI and Relationship Connector

Introduction

The IT registry Ci and Relationship Connector will add, update, delete, search or iterate CIs (Configuration Items) and the Relationships between them. For performing all listed operations, this Connector works directly with the IT Registry database. By using the **Artifact Type** parameter in the configuration panel of the Connector, you can specify whether its operation will be performed on Configuration Items or Relationships. Furthermore, you do not need to know the exact name of the artifact's class, and can directly discover the ones supported by the CDM (by pressing the **Select...** button in the configuration panel).

As with the IdML CI and Relationship Connector, the CDM meta-data can be retrieved from the local jar file or by connecting to a remote IT Registry system. You can choose between these options in the Advanced section of the Connector's configuration panel. Other usability features are the ability to test the connection to the remote IT Registry system (if it is being used). For the IT Registry case, the user can also provide information how to connect to IT Registry (for example, JDBC URL, Driver, username and password). If these fields are left blank the default values specified in the `etc/it_registry.properties` file will be used.

The selected CDM source also affects what attributes are displayed when querying the schema of the Output Map and Input Map of the Connector. For instance, if the IT Registry CDM is used, when listing a CI's attributes, you will get not only the specific attributes of its class (as is when the JAR meta-data is used), but also those of its parent classes. This of course leads to a bit slower response than when the JAR CDM is used. This is most notable when listing the Relationship types in the Connector's configuration panel. With the IT Registry CDM, you will be able to see additional information, specifying

which classes of CIs can act as sources and which as targets of the chosen Relationship. This can be very useful if you are unaware of the restrictions for the needed Relationship, but is a fairly slow operation which requires much more time than if the JAR CDM is used (then only the relationship types will be listed, without the class restrictions).

The naming rules limitation is also valid for this Connector. You are able to see all attributes of the chosen artifact class but can not determine which ones are part of a naming rule (also known as identifying attributes) and what rule exactly. Thus, in order to satisfy the required attributes of a CI, you need to find information for them in the CDM.

This Connector accepts a **Book Name** parameter, and will look up a different book, depending on its value. The name of the used book can also be overridden at runtime by the `$itRegistryBookName` attribute.

All GUIDs handled by this Connector should be wrapped as `com.ibm.di.fc.itregistry.ConfigurationItemId` objects. This is valid for both GUIDs that are passed to the Connector and those that are returned by it. This way, you can still map them in AssemblyLines, while not seeing the specific GUID format.

When the IT registry Ci and Relationship Connector is in CallReply mode, it is capable of registering/modifying both Configuration Items and Relationships along with their attributes. However, when users register CIs they should provide only identifying attributes, as IT registry does not support non-identifying ones in its current release.

For working properly in CallReply mode, this Connector depends upon the Init IT registry FC for details about the operation it should perform. This data is accessed through a shared book and includes a flag determining whether the Connector should perform a Refresh operation and a timestamp, used for distinguishing which artifacts should be "refreshed" and which not. See section “Open IdML Function Component” on page 500 for details on the Refresh operation.

For this mode, an important parameter which is provided in the Connector’s Output Map is the `$operation` attribute. It determines what operation will be performed with the specified CI/Relationship, when it is registered to the IT Registry database. It can be either added to the IT Registry database (the CREATE operation), updated (MODIFY), or removed (DELETE). These values – CREATE, MODIFY and DELETE (case insensitive) can be set in the `$operation` attribute. Note that if the used book is opened as a Refresh one (see the “Init IT Registry Function Component” on page 510 for information) only the CREATE operation is supported and passing any other value will cause an Exception. If no value is specified for the `$operation` attribute the CREATE value will be used by default.

The other option for setting the IT Registry operation is to pass a delta enabled work entry to the Connector. Since the IT Registry Ci and Relationship Connector is "delta aware", it will interpret the delta operation set to the entry and map its value to the IdML operations. The mapping is fairly straightforward:

Table 74. Delta codes to IdML Operations mapping

Delta operation	IdML operation
ADD (`Entry.OP_ADD`)	CREATE
MODIFY (`Entry.OP_MOD`)	MODIFY
DELETE (`Entry.OP_DEL`)	DELETE

Keep in mind that the provided delta operation will ALWAYS override the value of the `$operation` attribute.

Note: Due to limitations of the current version of the IT Registry, the UPDATE operation is not supported by the IT Registry Ci and Relationship Connector. If you try to provide it, an exception will be thrown.

When you query the schema of the Output Map of this Connector, the attributes of the chosen CI/Relationship class will be listed.

Querying the schema of a Relationship using the IT Registry Ci and Relationship Connector will return only two attributes:

- `source` - this is the GUID of the Managed Element that is the source of the Relationship
- `target` - this is the GUID of the CI that is the target of the Relationship

If any of these attributes is not provided, an Exception will be thrown.

To delete a Configuration Item, a set of identifying attributes must be passed to this Connector along with a DELETE value for the `$operation` attribute. If the Configuration Item is owned by several `ManagementSoftwareSystem`-s, it will not be deleted from the database. Instead, only the associations between the CI and the `ManagementSoftwareSystem` will be removed, thus releasing it from the MSS context. Similarly, to release/delete a Relationship for a specified MSS, the MSS's GUID and the GUIDs of the source and target CIs are required.

To facilitate reading of CIs, Iterator mode is provided. When the IT registry Ci and Relationship Connector is in this mode, it returns Configuration Items based on three separate criteria:

- *Attribute Filter* – a set of key-value pairs, specifying a filter for limiting the returned CIs. Each pair represents a CDM attribute and its required value. Thus, only CIs which have these attributes and their values will be returned by the Connector. If the filter is empty, all CIs will be fetched. The key-value pairs should be separated by commas and should use the '=' assignment operator. For example: `cdm:Model=T61p`, `cdm:Manufacturer=IBM`, `cdm:Fqdn=www.ibm.com`.
- *Date Filter* – a text field expecting a valid date in short format (for example, for US local , it should be MM/DD/YY). An empty Date Filter will throw error.

  Note: Either the Attribute Filter or Date Filter can be used by a single Connector. By default the Attribute Filter is enabled, but you can switch to Date Filter by checking the **Enable Date Filter** option.
- *MSS Name* – a text field accepting the name of an MSS (or a combination of its hostname, manufacturer and product name). By default it is left blank, meaning that all Configuration Items and Relationships which match the other filtering criteria will be returned, without checking the MSS-s they belong to. However, if an MSS name is provided in this field, only artifacts of that Management Software System will be returned. Instead of entering the MSS name themselves, users can list all MSSs present in the IT registry database (by pressing the **Get MSS Name** button) and directly select the needed one.

  Note: This parameter can be used independently of the Attribute Filter and Date Filter.

Lookup mode can be used to return a Configuration Item with all of its identifying attributes as stored in the IT Registry database. The search is performed using the conditions provided in the Link criteria of the Connector, If they are not specific enough, multiple Configuration Items can be returned. In this case, you should enable the **On Multiple Entries** hook and add some custom logic for handling the situation.

When specifying the Link Criteria, bear in mind to use only AND logical operations between the conditions and to rely only on the EQUALS operator.(for example, `cdm:Model=T61p` AND `cdm:Manufacturer=IBM` is a valid criteria). As in Iterator mode, you can further limit the returned CIs by providing a value for the MSS Name filter.

Schema

Output Schema:

$itRegistryBookName
This attribute can be used to override the name of the book used by this Component.

$operation
This attribute determines the IT Registry operation that will be performed with the specified CI/Relationship. If no value is specified, "create" will be used. Also, if a delta tagged entry is passed to the Connector, its delta operation will override the one specified by the `$operation` attribute.

CDM attributes of the CI/Relationship
Their names are retrieved from the used CDM, in order to create a valid CI.

$mssGuid
This attribute holds the MSS's GUID which is used for registering the CI/Relationship (this way the artifact will be associated with that Management Software System). It is wrapped as a `com.ibm.di.fc.itregistry.ConfigurationItemId` object.

Input Schema:

$guid This attribute contains the GUID given to the created Configuration Item, wrapped as a `com.ibm.di.fc.itregistry.ConfigurationItemId` object. Its value will be generated only when creating a Configuration Item, while for Relationships it will be *null*.

$managementSoftwareSystem
This attribute contains the details of the Managed Software System associated with the current CI. It is an Array of HashMap.

CDM attributes of the CI/Relationship
Their names are retrieved from the used IT Registry, when iterating over or looking up Configuration Items (in Iterator and Lookup modes).

Configuration

The IT Registry Ci and Relationship Connector uses the following parameters:

Artifact Type
Drop-down list; determines the type of artifact that this Connector will add to the IT Registry database.

Class Type
The type of Configuration Item or Relationship that will be created. You can use the **Select...** button to enter one of the pre-defined types.

Book Name
The name of the IT Registry book that will be used by this Connector. If left blank, the default IT Registry book will be used.

MSS Name
The name of Management Software System as present in the IT Registry database. If it is not present a combination of Manufacturer Name, Product Name and Host Name is displayed. You can use the **Select...** button to enter one of the pre-defined names.

Use IT Registry for CDM
Determines whether the FC will rely on IT Registry for the CDM meta data. You can use the **Get CDM version** button to check the version of the CDM provided by IT Registry or JAR file

JDBC URL
The JDBC URL used for connecting to the IT Registry database. Once you have provided all JDBC parameters, you can use the **Test Connection** button to test the JDBC connection to the IT Registry database.

JDBC Driver
The database driver used for connecting to the IT Registry database.

Username
The username used when connecting to the IT Registry database.

Password
The password used when connecting to the IT Registry database.

Enable Date Filter
This checkbox will determine whether **Date Filter** or **Attribute Filter** will be used for fetching CI from the IT Registry database. If checked, the Date Filter will be enabled.

Date Filter
Only Configuration Items for which the modification date is more recent than the filter will be returned. The format of the filter is (M/D/yy h:mm a).

Attribute Filter
This is the list of identifying attributes used for filtering the returned Configuration Items. Enter naming attributes for CIs or source and target attributes for Relationships. The attributes should be separated by ",".

The it_registry.properties file

The IdML Components, Open IdML Function Component and IdML Ci and Relationship Connector, need information how to connect to a IT Registry system, in order to retrieve the needed CDM meta-data. The same information is needed for the IT Registry Components, Init IT Registry Function Component and IT Registry Ci and Relationship Connector, which use it for registering information in the IT Registry database, as well as for retrieving CDM meta-data. Therefore, all of these Components have configuration fields in their panels, where you can specify the IT Registry information.

However, since in most cases you rely on only one IT Registry system most of the time, it is easier if there is a common place for this data. Therefore, Tivoli Directory Integrator provides a properties file, *TDI_solution_dir*/etc/it_registry.properties in which you can place the JDBC URL, JDBC Driver, Username and Password properties and they will be used by all IdML and IT Registry Components as default values.

The format of the it_registry.properties file is:

```
it_registry.jdbcUrl=
it_registry.jdbcDriver=
it_registry.dbUsername=
it_registry.dbPassword=
```

If different values are needed for any of these properties, you can either edit the it_registry.properties file (applying the change for all of the Components) or set the new values locally in the configuration panels of those Components needing them.

Examples

Steps to create a CI\Relationship using the IT Registry suite::

1. Add an Init IT registry FC. It will register the required MSS in the IT Registry database. Configure this Init IT registry FC:
   - Specify manually the **CDM version** used by this Component or use button **Get CDM version**.
   - Map the CDM attributes needed in the Output Map of the FC and supply values for them. Note that either cdm:MSSName or cdm:Hostname + cdm:Manufacturer + cdm:ProductName must be provided. To discover their names, click the **Connect** button in the Output Map of the FC.
   - Map the $mssGuid attribute in the Input Map of the FC.

- Optionally, specify a **Book Name** in the Advanced section of the configuration panel (if left blank the default book will be used).

2. Add an IT registry Ci and Relationship Connector in *CallReply* mode.
3. Configure this IT registry Ci and Relationship Connector:
   - In its configuration panel, select the **Artifact Type** (either CI or Relationship).
   - Next, select the **Class Type** for the artifact. Use the **Select...** button to list all supported class types.
   - Enter the same **Book Name** as specified for Init IT registry FC (or leave it blank if you have not specified a **Book Name** for the Init IT registry FC).
   - Map the `$mssGuid` returned by the Init IT registry FC to the Output Map of the IT registry Connector.
   - Map the CDM attributes needed in the Output Map of the FC and supply values for them. To discover their names, click the **Connect** button in the Output Map of the Connector.
   - Specify a valid value for the `$operation` attribute. The supported values are CREATE, MODIFY and DELETE (case insensitive). If the chosen operation is CREATE, the CI\Relationship will be registered in the IT Registry database.
   - If a CI is added, map the **$guid** attribute from the Input Map (for Relationships this attribute is null). This attribute can be used by another IT registry Ci and Relationship Connector to register a Relationship.
   - If a Relationship is added, make sure to map the source and target attributes in the Output Map of the Connector and provide the GUIDs of other registered CIs for their values (they can be taken from the **$guid** attributes).

Steps to create a CI\Relationship using the IdML suite::

(Only deviations from the steps for the IT registry suite are mentioned.)

1. Instead of the Init IT registry FC use the Open IDML FC to create an IdML File.
2. The Open IdML FC shares information with the IdML Connector through the IdML book. Therefore, provide the same **Book Name** for both Components.
3. The steps for configuring both Components are very similar.
4. Since no `$mssGuid` is present in the Input Map of the Open IdML FC, no mapping of this attribute is required.
5. The usage of the `$operation` attribute and the CDM attributes is the same.
6. The output of the IdML Ci and Relationship Connector is an `$id` attribute (as opposed to `$guid`) – a unique identifier of the artifact in the IdML Book. It should be used the same way as `$guid`-s – mapped to other IdML Ci and Relationship Connectors to create Relationships.

Note: For detailed instructions how to transform an AssemblyLine relying on the IdML Components into one using the IT Registry Components, see the IdML example.

IT Registry database setup

As mentioned before, the CDM Components (both the IdML and IT Registry suites) rely on two different sources for the CDM meta-data – a local JAR file and a remote IT Registry system. For advanced users Tivoli Directory Integrator provides a third alternative, namely setting up a local IT Registry database. Its main purpose is to provide you with a local copy of IT Registry's CDM meta-data definitions, so that a connection to the remote IT Registry system is not needed constantly. However, after its initial setup the local IT Registry is fully operational and can also be used for registering Configuration Items and Relationships. For more information about IT Registry 1.1 and its usage, see “Overview” on page 497.

All the files needed for the setup are shipped along with Tivoli Directory Integrator, on the installation DVDs and eAssemblies. The only prerequisites that you need to keep in mind are:

Operating system

The setup scripts for creating the IT Registry database are supported on the platforms supported by Maximo® (see the list below).

Note: Since i5/OS and z/OS platforms are not supported, the IT Registry setup scripts are not included on their media. If you need a local IT Registry database, it must use one of the following operating systems:

- AIX 5.3 - iSeries / pSeries and AIX 5L 6.1 - iSeries / System p
- HP-UX 11i v2 - PA-RISC and HP-UX 11i v3 - PA-RISC
- RHEL 4 - x86-32, RHEL 4 - zSeries® /System z and RHEL 5 - zSeries /System z
- Solaris 9 – SPARC and Solaris 10 – SPARC
- SUSE (SLES) 9.0 - zSeries /System z and SUSE (SLES) 10.0 - zSeries /System z
- Windows Server 2003 Datacenter Edition (Optional) - x86-32 and x86-64, Windows Server 2003 Enterprise Edition - x86-32 and x86-64, Windows Server 2003 Standard Edition - x86-32 and x86-64, Windows Server 2003 Standard x64 Edition - x86-32 and x86-64, Windows Vista - x86-32 and x86-64, Windows XP Professional - x86-32 and x86-64

Database

The IT Registry version 1.1 will support the databases supported by Maximo; setup scripts are only provided for those databases. Their names and version are listed below:

- MS SQL,version 2005 Standard Edition or Enterprise Editions (on Windows only)
- IBM DB2 8.2 + FP 7/1/07 or newer, DB2 UDB ESE 9.1 + FP 7/1/07 or newer
- Oracle 9i v2, Oracle 10 Rel1, Oracle 10 Rel2 and Oracle 11i for xSeries Linux

Steps to set up a local IT Registry database (assuming a suitable database is already installed)::

1. Open the Tivoli Directory Integrator install DVD and copy the `it_registry_dbscript.zip` archive. It contains everything needed to set up the IT Registry database.
2. Extract the archive to a suitable location, for example `C:\temp` and view its contents.
3. There you should find three archives – `disDb2Setup.zip`, `disMssqlSetup.zip` and `disOracleSetup.zip`, corresponding to the supported database types.
4. Extract the archive of the database you are planning to use, for example `disDb2Setup.zip`. It contains two setup scripts - `disDb2Setup.bat` (for Windows systems) and `disDb2Setup.sh` (for UNIX/Linux systems) and a folder named `/sql` that holds the SQL statement files containing the schema of the database and its CDM meta-data definitions (for example, `disDb2Views.sql`, `disDb2Schema.sql`).
5. Run the disSetupDb2.bat(.sh) from the DB2 command window (`db2cmd.exe`, found in the BIN directory in the install folder of DB2). You will need to provide the name of the database (if it does not exist, it will be created), username and password. The command should look like:

   ```
   disSetupDb2.bat –d db_name –u username –p password
   ```

If a problem occurs during the population of the IT Registry database, you may check the log files created in the `logs/` subfolder of the script's directory.

Troubleshooting

Locked books

If an AssemblyLine using the IdML Components crashes and they cannot shutdown normally (their `terminate()` method is not invoked), this could cause a permanent locking of the book they were using. Subsequent calls of this AL will display an error message that the needed book is already in use. To fix this you can either provide a new book name for the IdML Components or restart the Tivoli Directory Integrator server.

Database errors

While working with the IT Registry Components you may run into lower level database errors like:

```
com.ibm.tivoli.namereconciliation.common.NrsDatabaseException: 3001. An unexpected database system error has occu
```

The displayed message is very generic, so to be able to further debug the issue, you need to enable the IT Registry logging and tracing. This is achieved by performing the following steps:

1. Create a logging properties file for configuring the logging activities. This file follows the standard format used for setting up logging and specifies the required logging level, handlers and handler settings. Here is an example file:

```
############################################################
#   Default Logging Configuration File
#
# "handlers" specifies a comma separated list of log Handler
# classes.  These handlers will be installed during VM startup.
handlers= com.ibm.tivoli.dataintegration.common.logging.DISLogFileHandler, com.ibm.tivoli.dataintegration.com
# Default global logging level. The valid logging levels are: SEVERE (highest value), WARNING, INFO, CONFIG,
.level= FINEST
# DIS Log File Handler
# default file output is in user's home directory.
com.ibm.tivoli.dataintegration.common.logging.DISLogFileHandler.pattern = logs/dis%u.log
com.ibm.tivoli.dataintegration.common.logging.DISLogFileHandler.limit = 5000000
com.ibm.tivoli.dataintegration.common.logging.DISLogFileHandler.count = 1000
com.ibm.tivoli.dataintegration.common.logging.DISLogFileHandler.formatter = java.util.logging.SimpleFormatter
# DIS Trace File Handler
# default file output is in user's home directory.
com.ibm.tivoli.dataintegration.common.logging.DISTraceFileHandler.pattern = logs/dis%u.trace
com.ibm.tivoli.dataintegration.common.logging.DISTraceFileHandler.limit = 5000000
com.ibm.tivoli.dataintegration.common.logging.DISTraceFileHandler.count = 1000
com.ibm.tivoli.dataintegration.common.logging.DISTraceFileHandler.formatter = java.util.logging.SimpleFormatt
# Facility specific properties.
# Provides extra control for each logger.
#com.ibm.tivoli.namereconciliation.api.level = FINE
#com.ibm.tivoli.namereconciliation.service.level = FINE
#com.ibm.tivoli.namereconciliation.service.plugins.level = FINE
#com.ibm.tivoli.namereconciliation.service.plugins.cdm.level = FINE
#com.ibm.tivoli.datacleanser.level = FINE
#com.ibm.tivoli.dataintegration.metadata.level = FINE
```

   Please notice the paths in bold – **logs/dis%u/log** and **logs/dis%u.trace**. They determine where the IT Registry log and trace files will be stored and the file name format. Also, this configuration causes all occurring events to be logged since its logging level is set to FINEST. This is required for discovering some database errors, which are logged only at the lowest level. For more details on Java logging and the configuration file see http://www.oracle.com/technology/pub/articles/hunter_logging.html.

   Save the file as `dis.logging.properties` and place it in the solution directory of Tivoli Directory Integrator.

2. Modify the ibmdisrv.bat script to include the created file as a logging configuration file when TDI starts. For Windows systems make the following change (where the text in **bold** is what needs to be added):

```
rem Take the supported env variables and pass them to Java program
set LOG_4J=-Dlog4j.configuration="file:etc\log4j.properties"
set DIS_LOG=-Djava.util.logging.config.file=dis.logging.properties
set ENV_VARIABLES=%LOG_4J% %DIS_LOG%
"%TDI_JAVA_PROGRAM%" -classpath "%TDI_HOME_DIR%\IDILoader.jar" %ENV_VARIABLES% com.ibm.di.loader.ServerLauncher %*
```

   For Linux/UNIX systems the change is very similar, in ibmdisrv.sh make the changes outlined in **bold**:

```
# Log4j configuration file
LOG_4J=-Dlog4j.configuration=file:etc/log4j.properties
DIS_LOG=-Djava.util.logging.config.file=dis.logging.properties
"$TDI_JAVA_PROGRAM" $TDI_MIXEDMODE_FLAG -cp "$TDI_HOME_DIR/IDILoader.jar" "$LOG_4J" "$DIS_LOG" com.ibm.di.loader.ServerLauncher "
```

The path to `dis.logging.properties` can vary depending on the location where you placed the file. If it is in TDI's solution directory, only its name is needed.

3. Start Tivoli Directory Integrator. The log/trace files should appear in the `/logs` folder in the solution directory.

Chapter 7. Script languages

With this version of IBM Tivoli Directory Integrator the only script language available is JavaScript, implemented by means of the IBM JavaScript Engine (IBMJS), with Rhino compatibility extensions. If you previously have used VBScript, PerlScript or even BeanShell, you will need to convert this to JavaScript.

If your JavaScript code relies on particular extensions to the Rhino implementation, you may find that you have to change your code to equivalent IBM JavaScript Engine functionality. For example, if you had used the `org.mozilla.javascript.Synchronizer` class for synchronization, then this will not work anymore since the constructor for the class requires a `org.mozilla.javascript.Scriptable` object, which is something that belongs to the Rhino Script Engine. IBMJS provides the **synchronized** keyword; see "Main object" on page 526 for an example on how to use it.

JavaScript

There are certain issues you might want to consider when using JavaScript. These are:

- "Java + Script ≠ JavaScript" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.
- "Char/String data in Java vs. JavaScript Strings" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.
- "Java and JavaScript"

Java and JavaScript

In JavaScript you can access Java objects. This is very useful, because all the IBM Tivoli Directory Integrator internal objects are Java objects.

However, there is a pitfall when some of the Java Objects have methods with names that are reserved words or operators in JavaScript. In these cases, the JavaScript interpreter tries to process the reserved word instead of calling the Java method.

Such an example can be found with the **java.io.File** class which has a delete method. **delete** is also a JavaScript operator, so the following call fails:

```
var myFile = java.io.File("file.txt"); myFile.delete();
```

Instead, you can do one of the following calls:

- `myFile['delete']();`
  This exploits the fact that you can access the Java methods as array elements.
- `system.deleteFile("file.txt");`
  This works well, because the system library has a **deleteFile** method.

Chapter 8. Objects

The objects discussed in this chapter are fully documented in the Javadocs in the *root_directory*/docs/api directory of your installation. Check the Javadocs for the available methods; you can view the JavaDocs by selecting the **Help -> Welcome** screen, **JavaDocs** link in the Config Editor.

The AssemblyLine Connector object

The AssemblyLine Connector object is a wrapper that provides additional functionality to the Connector Interface. The Connector Interface can be accessed from the AssemblyLine Connector as the connector object.

Note: In addition to using the name of the AssemblyLine Connector, you can always refer to the currently executing AssemblyLine Connector object with the name "thisConnector" in your JavaScript code.

The AssemblyLine Connector is the Connector calling the hook functions you define in the AssemblyLine and is also the Connector that performs the attribute mapping. Each AssemblyLine Connector in the AssemblyLine is given a name that is automatically available in your scripts as that name. If you name an AssemblyLine Connector **ldap**, that name is also used as the **script object name**. Make sure you name your Connectors in a way that can be used as a JavaScript variable. Typically, you must avoid using whitespace and special characters.

The AssemblyLine Connector has methods and properties described in the com.ibm.di.server.AssemblyLineComponent.

The attribute object

An attribute object is usually contained in Entry objects. An attribute is a named object with associated values. Each value in the attribute corresponds to a Java object of some type. Attribute names are not case-sensitive, and cannot contain a slash (/) as part of the name. Remember that some of the Connectors for example, those accessing a database, might consider other characters as unsuitable. If you can, try to stick to alphanumeric characters in attribute names.

If the attribute was populated with Connector values by the attribute map, the values are of the same datatype that the Connector supplied.

For more information, see the Javadocs material included in the installation package (the com.ibm.di.entry.Attribute class).

Examples

Creating a new attribute object

```
var attr =  system.newAttribute("AttributeName");
```

This example creates an attribute object with name **AttributeName** and assigns it to the **attr** variable. Note that upon initial creation, the attribute holds no value. Now, through the **attr** variable you can access and interact with the newly created attribute.

Adding values to an attribute

```
attr.addValue("value 1");
attr.addValue("value 2");
```

This example adds the string values **"value 1"** and **"value 2"** to the **attr** attribute, thereby creating a multiple values attribute. Consecutive calls to **addValue(obj)** add values in the same order in the attribute.

Scanning attribute's values

```
var values = attr.getValues();
for (i=0; i<values.length; i++) {
  task.logmsg("Value " + i + " –> " + values[i]);
}
```

This example processes any attribute object, whether it holds single or multiple values. In reality, there is no difference between single and multiple-value attributes. Every attribute can hold zero, one or more values. A single-value attribute is therefore merely an underloaded multiple-values attribute.

See also

"The Entry object."

The Connector Interface object

The Connector Interface object is obtained either by loading a Connector Interface explicitly (system.loadConnector) or by retrieving the named AssemblyLine Connectors's .connector (myConnector.connector). When writing scripts in an AssemblyLine, you usually use the AssemblyLine Connector object that gives you access to another set of methods.

The Connector Interface is fully described as Connector in the Javadocs. For more information, see the Javadocs material included in the installation package (`com.ibm.di.connector.Connector`).

Methods

Some of the often-used methods include:

getNextEntry()
: Returns the next input entry.

putEntry (entry)
: Adds or inserts an **entry**.

modEntry (entry, search)
: Modify entry identified by **search** with contents of **entry**.

deleteEntry (entry, search)
: Deletes the **entry** identified by **search**.

findEntry (search)
: Searches for an entry identified by **search**. If no entries are found, a **null** value is returned.

findEntry (attribute, value)
: Performs a search using `"attribute equals value"` and returns the entry found. If no entries or more than one entry is found a null value is returned.

The Entry object

The Entry object is used by AssemblyLines. The Entry object is a Java object that holds attributes and properties. Attributes in turn contain any number of values. Properties contain a single value. For more information, see the Javadocs material included in the installation package (`com.ibm.di.entry.Entry`).

Global Entry instances available in scripting

conn The local storage object in Connectors in an AssemblyLine. It only exists during the Attribute Mapping phase of the Connector's life. See "Attribute Mapping" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.

work The data container object of the AssemblyLine. It is therefore accessible in every Connector from the AssemblyLine.

current
Available only in Connectors in Update and Delta mode. Stores the Entry that the Connector read from the data source to determine whether this is an Add or Modify operation.

error An Entry object that holds error status information in the following attributes:

status **(java.lang.String)**
ok if there is no exception thrown (in this case, this is the error's only attribute). **fail** if an exception is thrown, when the following attributes are also available:

exception **(java.lang.Exception)**
The **java.lang.Exception** (or some its successor class) object that is thrown

class **(java.lang.String)**
The name of the exception class (**java.lang.Exception** or some of its successors)

message **(java.lang.String)**
The error message of the exception

operation **(java.lang.String)**
The operation type of the Connector (for example, AddOnly, Update, Lookup, Iterator and so forth)

connectorname **(java.lang.String)**
The name of the Connector whose Hook is being called

thisScriptObject
An Entry object with the following Attributes:

AssemblyLine
Name of the executing AssemblyLine, or null if not available.

Component
Name of the executing Component, or null if not applicable.

HookName
Translated name of the Hook that is being executed, or null if not executing a Hook.

InternalHookName
Internal name of the Hook that is being executed, or null if not executing a Hook.

Attribute
The name of the Attribute being mapped, or null if not mapping an Attribute.

Map The String "Input" or "Output", if mapping an Attribute, null otherwise.

Function
The name of the function being called in a ScriptConnector, scriptParser or Scripted FC, otherwise null.

The simplest way to use this in a script would be for example,

```
task.logmsg("We are now executing " + thisScriptObject)
```

This will just print all available Attributes in the Entry.

See also

"The attribute object" on page 523.

The FTP object

The FTP object is available as a scriptable object. This object is useful when the FTP Client Connector does not provide the required functionality. See the full documentation in the Javadocs for `com.ibm.di.protocols.FTPBean`.

Example

```
var ftp = system.getFTP();

if ( ! ftp.connect ("ftpserver", "username",
  "password") )
{
 task.logmsg ("Connect failed: " +
  ftp.getLastError());
}

ftp.cd ("/home/user1");
var list = ftp.dir();
while ( list.next() )
{
 if (list.getType() == 1)
  task.logmsg ("Directory: " +
   list.getName());
 else
  task.logmsg ("File: " + list.getName());
}

ftp.setBinary();
ftp.get ("remotefile", "c:\\localfile");
ftp.put ("c:\\localfile", "remotefile");
```

Main object

The **main** object is the top level thread (see Interface `RSInterface` in the Javadocs). This object has methods for manipulating AssemblyLine behavior. The most common methods are:

void dump(object)
: Dumps the object to the log file. If object is an**Entry** , Dumps the object to the log file; otherwise, just the class name and object.toString().

void logmsg (String loglevel, String msg)
: Alternative version of the `logmsg()` method, with a Log Level parameter. The legal values for Log Level are: "FATAL", "ERROR", "WARN", "INFO", "DEBUG", corresponding to the log levels available for log Appenders. Any unrecognized value is treated as "DEBUG".

startAL (name, initial-work-entry), startAL (name, runtime-provided-Connector), startAL (name, initial-work-entry, runtime-provided-Connector), startAL (name, java.util.Vector)
: Starts the AssemblyLine given by the **name** parameter. See also "IBM Tivoli Directory Integrator concepts – The AssemblyLine" in *IBM Tivoli Directory Integrator V7.1 Users Guide*.

If there is a need to synchronize between multiple AssemblyLines for some reason, you can take advantage of the synchronized keyword in the IBM Javascript engine. This can be used to synchronize on some common Object. For example, if the AssemblyLines are all running in the same Config Instance, they could synchronize on the `main` Object, like this:

```
synchronized(main) {
  task.logmsg("Inside the synchronized block")
 }
```

The Search (criteria) object

The **Search (criteria)** object is used by AssemblyLines and Connectors to specify a generic search criteria. See `com.ibm.di.server.SearchCriteria` in the Javadocs. It is up to each Connector how to interpret and translate the search criteria into a Connector specific search. The search criteria is a very simple multi-valued object where each value specifies an attribute, operand, and a value.

Operands

The following operands have been defined for use with the criteria objects.

= Equals

~ Contains

^ Starts with

$ Ends with

! Not equals

Example

```
for ( i = 0; i < search.size(); i++ ) {
 var sc = search.getCriteria ( i );
 task.logmsg ( sc.name + sc.match + sc.value );
}
```

The shellCommand object

The **shellCommand** object contains the results from a command line process.

On Microsoft Windows platforms, the shell command starts, but you cannot get output or status from the shell command. See `com.ibm.di.function.ExecuteCommand` in the Javadocs for available methods.

For example:

```
var cmd = system.shellCommand ("/bin/ls -l");
if ( cmd.failed() ) {
 task.logmsg ( "Command failed: " + cmd.getError());
} else {
 task.logmsg ( cmd.getOutputBuffer() );
}
```

The status object

The **status** object contains information about an AssemblyLine's Connectors and error codes. It is a synonym to **task.getStats()**

The system object

The **system** object is available as a scriptable object in all scripting contexts and provides a basic set of functions. The Java object is `com.ibm.di.function.UserFunctions`, but linked to the Script object **system**. You can find a complete list of the methods by looking at the Javadocs.

The task object

The **task** object is an instance of class that implements `com.ibm.di.server.TaskInterface` and represents the current thread of execution:

- For AssemblyLines, this is the AssemblyLine thread where you can access AssemblyLine specific information and methods. See class `com.ibm.di.server.AssemblyLine` in the Javadocs.

The COMProxy object

The COMProxy object allows you to call COM Automation components from Java. Java Native Interface (JNI) makes native calls into the COM and Win32 libraries. COMProxy makes use of Object Linking and Embedding (OLE) Automation under the wraps (also knows as late binding) to make calls to COM objects/interfaces. COMProxy also supports Moniker URLs. To obtain a handle to a COMProxy instance use the `system.createCOMInstance( )` method.

Notes:

1. Full documentation for the COMProxy is available in the Javadocs under the `com.ibm.di.automation` package.
2. The COMProxy object does not support ADSI calls.

The COMProxy object is described by way of an example; the Connector described here is a re-implementation in JavaScript of an older Outlook Connector written in VBScript. You can find the code in the *TDI_install_dir*`/examples/MSOutlook` directory.

This example shows how you can manipulate your Outlook Contacts using COMProxy. It is an example of an `ibmdi.scriptconnector`, and shows how you can create a script connector that supports add, iterate, update, lookup, and delete modes.

The script code is provided below if you would like to create your own script connector and input this data. The file *msoutlook.xml* is a Tivoli Directory Integrator Config file with the Connector already entered for you. If you open `msoutlook.xml` you will find a scriptconnector called **msoutlook** that contains this script information ("config"->"script").

You can also copy the MSOutlook.jar file to the *TDI_install_dir*`//jars/connectors` directory, after which it will appear in your list of available connectors to inherit from.

Example code

```
//
// This script implements all the necessary functions for accessing
// the Contacts register in MS Outlook.
// Assumes that the number of entries in contact folder is constant for the run

ol = system.createCOMInstance("Outlook.Application");

ns = COMProxy.call(ol,"GetNameSpace","MAPI");

contacts = COMProxy.call(ns.toObject(),"getDefaultFolder",10);

var item;
var counter = 0;
var oldstring="";

var decode="";

var outlookEntry = system.newEntry();

function selectEntries(){
 counter = 0;
}

function getNextEntry (){
 ol = system.createCOMInstance("Outlook.Application");

 ns = COMProxy.call(ol,"GetNameSpace","MAPI");

 contacts = COMProxy.call(ns.toObject(),"getDefaultFolder",10);

 items = COMProxy.call(contacts.toObject(),"Items");

 count = COMProxy.get(items.toObject(),"count");

 counter++;
 if(counter > count){
  result.setStatus(0);
  result.setMessage("End of input");
```

```
 }else{
  item = COMProxy.call(items.toObject(),"Item",counter);
  populateEntry();
 }
}

function findEntry (){
 flt = "[" + search.getFirstCriteriaName() + "] = " + search.getFirstCriteriaValue();
 items = COMProxy.call(contacts.toObject(),"Items");
 item = COMProxy.call(items.toObject(),"Find",flt);
 if (item == null){
  result.setStatus(0)
  result.setMessage("Not found" + "--->["+ flt + "]");
 }
 else
  populateEntry();
}

function modEntry (){
    populateItem();
    COMProxy.call(item.toObject(),"Save");
}

function deleteEntry (){
 COMProxy.call(item.toObject(),"Delete");
}

function putEntry (){
 items = COMProxy.call(contacts.toObject(),"Items");
 item = COMProxy.get(items.toObject(),"Add");
 if(item==null){
  result.setStatus(2)
  result.setMessage("Unabled to create item");
  return;
 }
 oldString = entry.getString("FullName");
 COMProxy.put(item.toObject(),"FileAs",oldString);
 populateItem();
 COMProxy.call(item.toObject(),"Save");
}

function populateEntry (){
 entry.setAttribute("FileAs", COMProxy.get(item.toObject(),"FileAs"));
 entry.setAttribute("FullName", COMProxy.get(item.toObject(),"FullName"));
 entry.setAttribute("Email1Address", COMProxy.get(item.toObject(),"Email1Address"));
 entry.setAttribute("Birthday", COMProxy.get(item.toObject(),"Birthday"));

 entry.setAttribute("BusinessAddress", COMProxy.get(item.toObject(),"BusinessAddress"));
 entry.setAttribute("BusinessTelephoneNumber", COMProxy.get(item.toObject(),"BusinessTelephoneNumber"));
 entry.setAttribute("BusinessFaxNumber", COMProxy.get(item.toObject(),"BusinessFaxNumber"));
 entry.setAttribute("CompanyName", COMProxy.get(item.toObject(),"CompanyName"));
 entry.setAttribute("JobTitle", COMProxy.get(item.toObject(),"JobTitle"));

 entry.setAttribute("HomeAddress", COMProxy.get(item.toObject(),"HomeAddress"));
 entry.setAttribute("HomeTelephoneNumber", COMProxy.get(item.toObject(),"HomeTelephoneNumber"));
 entry.setAttribute("HomeFaxNumber", COMProxy.get(item.toObject(),"HomeFaxNumber"));

 entry.setAttribute("MobileTelephoneNumber", COMProxy.get(item.toObject(),"MobileTelephoneNumber"));

 entry.setAttribute("Categories", COMProxy.get(item.toObject(),"Categories"));
 entry.setAttribute("LastModificationTime", COMProxy.get(item.toObject(),"LastModificationTime"));
        outlookEntry = entry.clone(entry);
}

function populateItem (){
 outlookEntry.merge(entry);
        COMProxy.put(item.toObject(),"FileAs", outlookEntry.getString("FileAs"));
 COMProxy.put(item.toObject(),"FullName", outlookEntry.getString("FullName"));
 COMProxy.put(item.toObject(),"Email1Address", outlookEntry.getString("Email1Address"));
 COMProxy.put(item.toObject(),"BusinessAddress", outlookEntry.getString("BusinessAddress"));
 COMProxy.put(item.toObject(),"BusinessTelephoneNumber", outlookEntry.getString("BusinessTelephoneNumber"));
 COMProxy.put(item.toObject(),"BusinessFaxNumber",outlookEntry.getString("BusinessFaxNumber"));
 COMProxy.put(item.toObject(),"JobTitle", outlookEntry.getString("JobTitle")) ;
 COMProxy.put(item.toObject(),"CompanyName", outlookEntry.getString("CompanyName")) ;
 COMProxy.put(item.toObject(),"HomeAddress",outlookEntry.getString("HomeAddress")) ;
 COMProxy.put(item.toObject(),"HomeTelephoneNumber", outlookEntry.getString("HomeTelephoneNumber")) ;
 COMProxy.put(item.toObject(),"HomeFaxNumber", outlookEntry.getString("HomeFaxNumber")) ;
 COMProxy.put(item.toObject(),"Categories", outlookEntry.getString("Categories"));
 if (outlookEntry.getString("Birthday")!=null && !outlookEntry.getString("Birthday").equals(" "))
  COMProxy.put(item.toObject(),"Birthday", outlookEntry.getString("Birthday"));
}
```

See also

"Script Connector" on page 255.

Appendix A. Password Synchronization plug-ins

The IBM Tivoli Directory Integrator provides an infrastructure and a number of ready-to-use components for implementing solutions that synchronize user passwords in heterogeneous software environments.

A password synchronization solution built with the IBM Tivoli Directory Integrator can intercept password changes on a number of systems. The intercepted changes can be directed back into:

- The same software systems, or
- A different set of software systems.

Synchronization is achieved through the IBM Tivoli Directory Integrator AssemblyLines, which can be configured to propagate the intercepted passwords to desired systems.

The components that make up a password synchronization solution are: Password Synchronizers, Password Stores, Connectors and AssemblyLines. The Password Synchronizers, Password Stores and Connectors are ready-to-use components included in the IBM Tivoli Directory Integrator. As a result, implementing the solution that intercepts the passwords and makes them accessible from IBM Tivoli Directory Integrator is achieved by deploying and configuring these components.

Note: These components are not available in the Tivoli Directory Integrator 7.1 General Purpose Edition.

The following sections describe the specialized password synchronization components that are currently available.

Password Synchronizers

Password Synchronizer for Windows XP/Vista
Intercepts the Windows login password change.

Password Synchronizer for IBM Tivoli Directory Server
Intercepts IBM Tivoli Directory Server password changes.

Password Synchronizer for Sun Directory Server
Intercepts Sun ONE Directory Server password changes.

Password Synchronizer for Domino
Intercepts changes of the HTTP password for Lotus Notes users.

Password Synchronizer for UNIX and Linux
Intercepts changes of UNIX and Linux user passwords.

Password Stores

LDAP Password Store
Provides the function necessary to store the intercepted user passwords in LDAP directory servers.

JMS Password Store
JMS Password Store (formally known as the MQ Everyplace Password Store) provides the functionality necessary to store intercepted user passwords in a JMS Provider's Queue from where any JMS client for example, Tivoli Directory Integrator) could read them.

Log Password Store
The Log Password Store is solely used to log any actions that a normal password store would take. This password store is useful for verifying that the Java Proxy and the native plug-ins are communicating correctly.

Specialized Connectors

JMS Password Store Connector

Provides the function necessary to retrieve password update messages from IBM WebSphere MQ Everyplace and send them to IBM Tivoli Directory Integrator.

Tivoli Identity Manager Integration

The *IBM Tivoli Directory Integrator V7.1 Password Synchronization Plug-ins Guide*also details the steps required for integration between Tivoli Identity Manager and the following Password Synchronizers:

- Sun Directory Server Password Synchronizer,
- IBM Directory Server Password Synchronizer,
- Windows Password Synchronizer, and
- Password Synchronizer for UNIX and Linux.

For more information about installing and configuring the IBM Password Synchronization plug-ins, please see the *IBM Tivoli Directory Integrator V7.1 Password Synchronization Plug-ins Guide*.

Appendix B. AssemblyLine Flow Diagrams

AssemblyLine Flow Diagrams

The AssemblyLine Flow diagrams show where in a given Component's execution state elements like Hooks, and Entry objects like Conn, Work, Current and Error are available.

The flow diagrams are available in the following PDF document: TDI_7.1_FlowDiagrams.pdf

Appendix C. Server API

Overview

The IBM Tivoli Directory Integrator 7.1 Server API provides a set of programming calls that can be used to develop solutions and interact with the IBM Tivoli Directory Integrator (TDI) Server locally and remotely. It also includes a management layer that exposes the Server API calls through the Java Management Extensions (JMX) interface.

The Server API includes calls that allow you to:

- Get information about the TDI Server
- Get information about components installed on the Server
- Read, Modify and Write configurations loaded by the Server
- Create and Load new configurations on the Server
- Start, Query and Stop AssemblyLines
- Cycle manually through AssemblyLines
- Register for and receive notifications for Server events
- Register for and receive AssemblyLines log messages

All calls can be invoked locally from the TDI Server JVM, and remotely from another JVM (on the local or a remote network machine), through RMI:

Local access
: This type access includes scripting in AssemblyLine hooks and also using the API from new components (Connectors, Function Components) implemented in Java and deployed on the Server.

Remote access:
: This type of access enables the implementation of solutions that remotely connect to TDI and manage processes within TDI or/and build business logic on top of TDI. It could be an application dedicated solely to TDI or an application that uses TDI to accomplish some of its goals.

A management layer of the Server API exposes the Server API calls through JMX. This provides for Server manageability and enables you to plug TDI into a managing infrastructure that speaks JMX. The JMX interface is accessible:

- Locally, as defined in the JMX 1.2 specification
- Remotely, through RMI as defined by the JMX Remote API 1.0 specification

The notifications issued by the Server API internal engine are also available as JMX notifications.

Remote access to the Server API (including the JMX Remote API) is secured by using SSL with client and server authentication.

The different methods that can be used to access the TDI Server API are depicted on the diagram below:

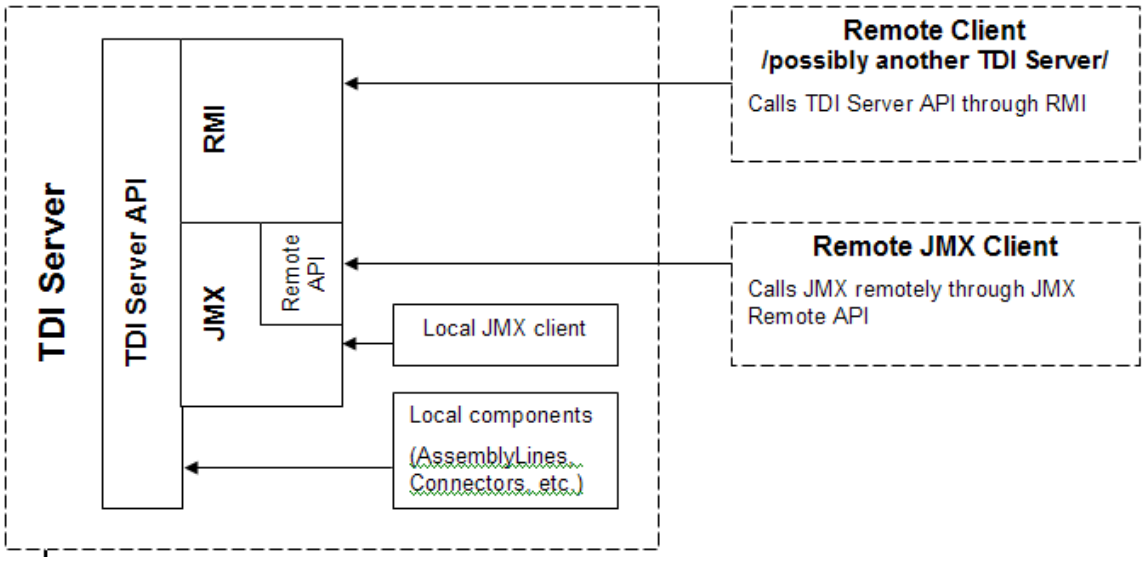


Figure 5.

Sample use case

In this sample scenario, a client (a stand-alone Java application, for example) needs to start an AssemblyLine on TDI Server. The client could use the Server API and access it remotely through the RMI interface, using the Server API RMI client library.

In accordance with the security model described in "Security" on page 537, the client will first create a session to the remote TDI Server using its own certificate or custom authentication. The Server will successfully authenticate the client if it has the client certificate in its trust store or custom authentication succeeds. If the authentication is successful the client will be provided with an object that represents an entry point for calling Server API methods. Using that object the client will invoke the call for starting an AssemblyLine passing parameters that specify which AssemblyLine needs to be started.

Before actually executing the method the Server API will check whether the client is authorized to execute that method – the identity of the client is determined through the client certificate used to establish the SSL channel or with provided credentials for the custom authentication. If the client is allowed to start this AssemblyLine the method will be executed and the AssemblyLine will be started; otherwise, the method will not be executed and an error (exception) will be sent back to the client indicating that it is not authorized to perform this operation.

Local and Remote Server API interfaces

The Server API provides two sets of interfaces: one for local use and one for remote use. Both sets of interfaces provide the same calls and functionality, but reside in different Java packages.

The package `com.ibm.di.api.local` contains the interfaces for local access and `com.ibm.di.api.remote` contains the interfaces for remote access to the Server through RMI.

Detailed specification of the local and remote interfaces and their methods can be found in the JavaDoc documentation shipped with the TDI (in the `docs/api` folder under the root folder where TDI is installed).

All interfaces in the remote package extend `java.rmi.Remote` and all their methods throw `java.rmi.RemoteException`. The interfaces for local access on the other hand do not extend

java.rmi.Remote and their methods do not throw java.rmi.RemoteException which facilitates coding and is one of the reasons to have separate set of interfaces for local and remote access.

Server API structure

The structure of the local and remote interfaces is identical. The text below refers to the names of the Java interfaces only and is valid for the interfaces from both the local (com.ibm.di.api.local) and remote (com.ibm.di.api.remote) Server API Java packages.

The entry point to the Server API is the SessionFactory interface (com.ibm.di.api.local.SessionFactory for local use and com.ibm.di.api.remote.SessionFactory for remote use).

The SessionFactory interface provides two methods createSession() and createSession(Username, Password). They create an API session for the user/entity that calls it and returns an object of type Session. It is this Session object that provides further access to the calls of the Server API.

Through the Session object one can get Server information or stop the Server itself, existing Config Instances can be obtained or new Config Instances can be loaded and created from scratch. Some of the calls of the Session object will return other Server API objects – for example startConfigInstance(String aConfigUrl) will return a ConfigInstance object. The ConfigInstance object gives access to the configuration data structure, to AssemblyLines running in the Config Instance as well as calls for starting new AssemblyLines. Some of its calls will also return Server API objects. startAssemblyLine(String aAssemblyLineName), for example, returns an AssemblyLine object that you can use to query and perform different operations on the AssemblyLine.

To summarize, the Session object is the one that gives access to the hierarchy of Server API objects. All Server API calls are either invoked directly on the Session object or they are invoked on objects retrieved directly or indirectly through the Session object.

Security

Authentication is performed in the process of obtaining the Session object. Once obtained, all methods called on the Session object or on other Server API objects retrieved directly or indirectly through this *Session* object are executed under the identity of the user that obtained the Session object.

Authorization is performed on each method call. Before executing the requested call, the Server will determine whether the identity associated with the current session is authorized to execute that call.

The following authentication options are available:

SSL-based authentication (the mechanism available in TDI 6.0)

This option functions only when api.remote.ssl.client.auth.on=true (you will also need api.on=true, api.remote.on=true, api.remote.ssl.on=true.

The user is authorized as per the rights assigned to the SSL certificate user ID in the Server API User Registry.

Note: When SSL is used and the remote client application uses Server API listener objects, the client application must have its own certificate that is trusted by the TDI Server (this is analogous to the setup for SSL client authentication). If there is no client certificate trusted by the TDI Server, the listener objects will not work and the remote client application will not be able to receive notifications from the TDI Server.

Username/password based authentication

This option functions only when api.custom.authentication is set to a JavaScript authentication file.

This authentication method works regardless of whether SSL is used and whether SSL client authentication is used. The user is authorized as per the rights assigned to the username user in the Server API User Registry.

An example authentication hook Javascript file is available in order to demonstrate what the Javascript of an authentication hook looks like. This example Javascript can also be used as the basis of real-world TDI authentication hooks.

You can view an JavaScript example that demonstrates how an authentication hook can use an LDAP server (Tivoli Directory Server, Active Directory, etc.) for authenticating client request in the examples/auth_ldap TDI Server folder. The example file is called ldap_auth.js.

LDAP authentication
The TDI Server API provides support for LDAP Authentication. This allows customers to leverage their existing LDAP infrastructures which already hold their User IDs and Passwords.

In order to use LDAP authentication the appropriate properties must be configured in global.properties/solution.properties. These properties are described in the Administrator Guide.

Host-based authentication
This option functions only when `api.remote.ssl.on=false`. If so, then opening of Server API sessions without username/password supplied from all hosts specified by the `api.remote.nonssl.hosts` property are successfully authenticated and granted admin authority. The `api.remote.nonssl.hosts` property can be specified in the global.properties/ solution.properties files.

Note: It is strongly recommended that you use this authentication only for demo purposes, quick prototyping and in closed, trusted environments.

JAAS authentication
The Server API provides support for JAAS Authentication. If you already already have JAAS authentication modules, this allows you to use them with TDI.

In order to use JAAS authentication the appropriate properties must be configured in `global.properties` or `solution.properties` and the JAAS Logon should be installed.

Note: Tivoli Directory Integrator 7.1 does not configure any JAAS authentication modules. It relies on the understanding that you have such implemented and properly configured. Tivoli Directory Integrator can simply use them then.

Configuring the Server API

Configuring the Server API on the Server side includes specifying the relevant system properties in `global.properties` (or `solution.properties`) and configuring the User Registry file.

Configuring the Server API properties

The Server API engine is configured through a set of properties in the `global.properties` file (or `solution.properties` file, if a solution folder is used). Refer to the chapter on Security and TDI, section "Server API Access Security" in the *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide* for information on how to configure the Server API.

Setting up the User Registry

Refer to the "Security and TDI" chapter in the *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide* for information and examples of how to setup the User Registry, assign user roles and encrypt or decrypt the User Registry file.

Remote client configuration

This section describes what is necessary for a remote client that will use the remote Server API.

Prerequisites:

Java 6 or higher is required on the client side.

Configuring the client:

1. The following jar files must be included in the CLASSPATH of the remote side:
   - jars/common/diserverapi.jar
   - jars/common/diserverapirmi.jar
   - jars/3rdparty/others/log4j-1.2.15.jar
   - jars/common/miconfig.jar
   - jars/common/miserver.jar
   - jars/common/mmconfig.jar
   - jars/common/tdiresource.jar
   - jars/3rdparty/IBM/icu4j_4_2.jar
   - jars/3rdparty/IBM/ITLMToolkit.jar
   - jars/3rdparty/IBM/jlog.jar

   You can copy these jar files from the Tivoli Directory Integrator installation.
2. If custom non-TDI objects are used in the solution being implemented with the Server API (for example as Attribute values of an Entry that is transferred over the wire) the corresponding Java classes have to be available on the client side as well. These classes must be serializable and they have to be included in the CLASSPATH of the client JVM.

SSL configuration of the remote client

There are two options for configuring SSL on the remote client:

Using Server API specific SSL properties

When the Java System property `api.client.ssl.custom.properties.on` is set to `true`, then SSL is configured through the following TDI Server API-specific Java System properties:

- **api.client.keystore** – specifies the keystore file containing the client certificate
- **api.client.keystore.pass** – specifies the password of the keystore file specified by *api.client.keystore*
- **api.client.keystore.type** – specifies the type of the keystore file specified by *api.client.keystore*; optional property, if not specified the default keystore format for the JVM will be used
- **api.client.key.pass** – the password of the private key stored in keystore file specified by *api.client.keystore*; if this property is missing, the password specified by *api.client.keystore.pass* is used instead.
- **api.client.truststore** – specifies the keystore file containing the TDI Server public certificate.
- **api.client.truststore.pass** – specifies the password for the keystore file specified by *api.truststore*.
- **api.client.truststore.type** – specifies the type of the keystore file specified by *api.client.truststore*; optional property, if not specified the default keystore format for the JVM will be used

Using the Server API-specific SSL properties is convenient when your client application is using the standard Java SSL properties for configuration of another SSL channel used by the same application.

You can specify these properties as JVM arguments on the command line, for example:

```
java MyTDIServerAPIClientApp
    -Dapi.client.ssl.custom.properties.on=true
    -Dapi.client.truststore=C:\TDI\serverapi\testadmin.jks
    -Dapi.client.truststore.pass=administrator
    -Dapi.client.keystore=C:\TDI\serverapi\testadmin.jks
    -Dapi.client.keystore.pass=administrator
```

This example refers to the sample `testadmin.jks` keystore file shipped with Tivoli Directory Integrator. Note that it contains both the client private key and also the public key of the TDI Server, so it is used as both as a keystore and truststore.

Using the standard SSL Java System properties:

When the Java System property `api.client.ssl.custom.properties.on` is missing or when it is set to `false`, then the standard JSSE system properties are used for configuring the SSL channel. Follow the standard JSSE procedure for configuring the keystore and truststore used by the client application.

You can specify these properties as JVM arguments on the command line; for example:

```
java MyTDIServerAPIClientApp
-Djavax.net.ssl.keyStore=C:\TDI\serverapi\testadmin.jks
-Djavax.net.ssl.keyStorePassword=administrator
-Djavax.net.ssl.trustStore=C:\TDI\serverapi\testadmin.jks
-Djavax.net.ssl.trustStorePassword=administrator
```

Using the Server API

Creating a local Session

If you are writing Java code that will be executed in the TDI Server JVM (for example a new Connector, or a Java class that you will access through scripting) and you want to execute Server API calls, you'll need a local Server API session.

You can obtain a local Server API session by calling:

```
import com.ibm.di.api.APIEngine;
import com.ibm.di.api.local.Session;

...

Session session = APIEngine.getLocalSession();
```

`getLocalSession()` is a static method of the `com.ibm.di.api.APIEngine` class. It creates and returns a new `com.ibm.di.api.local.Session` object. This session returned has admin rights and can execute all Server API calls.

Access to the Server API in a scripting context

Users can get access to the Server API from a scripting context (for example from AssemblyLine hooks) by calling the `session` script object. TDI Server registers session objects by calling `com.ibm.di.api.APIEngine.getLocalSession()` method.

Creating a remote Session

A client application that uses the remote Server API would first need to connect to the TDI Server and obtain a Server API Session.

Use the following Java code to lookup the RMI SessionFactory object and obtain a Server API Session:

```
import com.ibm.di.api.remote.Session;
import com.ibm.di.api.remote.SessionFactory;

...

SessionFactory sessionFactory = (SessionFactory) Naming.lookup("rmi://<TDI_Server_host>:
 <TDI_Server_RMI_port>/SessionFactory");

Session session = sessionFactory.createSession();
```

You need to replace *TDI_Server_host* and *TDI_Server_RMI_port* with the host and the RMI port of the TDI Server; for example:

```
Naming.lookup("rmi://127.0.0.1:1099/SessionFactory")
```

The calls provided by the local and remote Session objects are identical. All examples below assume that you have already obtained a session and will operate in a remote context. In other words, the remote versions of the Server API interfaces will be used.

Working with Config Instances

The Config Instance represents a configuration loaded on the TDI Server and the associated Server object. Each AssemblyLine is running in the context of a Config Instance. Through a Config Instance you can query the configuration of AssemblyLines, Connectors, Parsers, Functional Components, start AssemblyLines, get access to running AssemblyLines and query their log files.

Getting access to running Config Instances

You can obtain access to all Config Instances running on the Tivoli Directory Integrator Server by executing the following piece of code:

```
ConfigInstance[] configInstances = session.getConfigInstances();
for (int i=0; i<configInstances.length; i++) {
// do something with configInstances[i]
}
```

The `getConfigInstances()` method will return an array with Config Instance Server API objects representing all Config Instances running on the Server.

Starting a Config Instance

In order to load a new configuration on the TDI Server you need to start a new Config Instance, pointing it to the XML configuration file:

```
ConfigInstance configInstance = session.startConfigInstance("testconfig.xml");
```

This loads the testconfig.xml configuration file and start a new Config Instance object associated with that configuration. Once you get that Config Instance object you can use it to change the configuration itself, start AssemblyLines or stop the Config Instance on the Server when you no longer need it.

If you need to start multiple configuration instances from a single configuration file (for example if you want to use a different set of properties for each instance), you must provide a unique Run Name for each instance:

```
ConfigInstance configInstance = session.startConfigInstance("testconfig.xml", true, null,
 "myrunname", "mystore=mynewstore.properties");
```

The above call will start a new configuration instance from the "testconfig.xml" file with a Run Name "myrunname". That Run Name will be used as the id of the configuration instance. Furthermore, the property store "mystore" of the instance will be redirected to load its contents from the "mynewstore.properties" file.

Stopping a Config Instance

Assuming that you have a reference to the Config Instance Server API object, you can stop the Config Instance by calling:

```
configInstance.stop();
```

For a reference to the Config Instance object, you have the following options:

- Keep that reference from where you started the Config Instance, that is, `configInstance = session.startConfigInstance("testconfig.xml");`
- Retrieve the Config Instance object through its Config ID by calling `session.getConfigInstance (String aConfigId)`. The Config ID is a unique identifier for each Config Instance running on the

Server. It is created by the Server API when the corresponding Server API Config Instance object is created. You can retrieve the Config ID through the Config Instance object by calling `configInstance.getConfigId()`.

- Iterate through all running Config Instances and find the one you need: `session.getConfigInstances()` will return an array of all running Config Instances.

Synchronizing Server API and Config Initialization

If a config instance has not fully loaded the configuration file when a server API call is made, it returns a null object. In Tivoli Directory Integrator 7.1, a time out interval is configurable by means of a property - **api.config.timeout** . This is set to 2 mins by default. An exception will be thrown if the config has not been loaded within this time interval.

Optional Config instance ID in a Config file

The Config Instance is representing a configuration loaded on the Tivoli Directory Integrator Server and the associated Server object. Each AssemblyLine is running in the context of a Config Instance. Through a Config Instance you can query the configuration of AssemblyLines, Connectors, Parsers, Functional Components, start AssemblyLines, get access to running AssemblyLines and query their log files.

Solution Name and Run Name - the Configuration Instance ID: When a configuration file is loaded by the Tivoli Directory Integrator Server, it becomes a running **configuration instance**. Each configuration instance has its own **configuration id**. No two configuration instances running at the same time are allowed to have the same configuration id (a configuration id uniquely identifies a running configuration instance within the Tivoli Directory Integrator Server).

When a configuration instance is started off a configuration file, the Tivoli Directory Integrator Server first checks if the configuration file has a defined **Solution Name** (a configuration field of the Solution Interface configuration screen). If the Solution Name is present and non-emtpy, the Server uses this name as the configuration instance id. If the Solution Name is missing or empty, the Tivoli Directory Integrator Server automatically generates a configuration id.

For example if a configuration file with an absolute file name "C:/IBM/TDI/configs/rs.xml" is loaded into the Tivoli Directory Integrator Server and the file has a Solution Name set to "mysoluname", then the id of the spawned configuration instance is "mysoluname". If the same configuration had no Solution Name defined, the configuration instance id would be something like "C__IBM_TDI_configs_rs.xml".

Note: The clients of the TDI Server API must perceive the automatically generated configuration instance ids as transparent entities – they must not try to guess how these ids are generated, because the algorithm is subject to change in the future. The only guarantee is that if a configuration instance once existed under some automatically generated configuration id, then certain artifacts such as tombstones and system logs can be accessed later using the same configuration id. There is no guarantee, however, that if the same configuration file is run again, the newly spawned configuration instance will have the same automatically generated id as before.

Generally if the client specifies nothing more than the path to the configuration file while starting a configuration instance, the configuration id is based solely on the attributes of the configuration file (the Solution Name, if any, or the absolute file name). As a result, if no additional information is provided, a configuration file can be loaded as a configuration instance only once (otherwise there is a conflict of configuration instance ids).

If it is necessary to load multiple configuration instances from the same configuration file, the client needs to provide a unique **Run Name** for each of the instances. If a run name is supplied when starting a configuration instance, that run name is used as the configuration instance id of the instance. Consequently a Run Name must not coincide with any of the ids of already running configuration instances.

Each Solution Name and each Run Name must be a valid file name on the platform on which the TDI Server is currently running. The reason for this restriction is that the configuration instance id (which

derives from the Solution Name or the Run Name) is used when storing certain configuration-instance-specific information, such as the System Logs. To avoid file system problems, TDI forbids the following symbols to appear inside a Run Name or a Solution Name: \ / : * " < > | ?

Notes:

1. If a configuration instance is started with a Solution Name that has any of the above symbols in it, the TDI Server will automatically replace that problem symbols with underscores and will log a warning. If a client attempts to start a configuration instance with a Run Name that contains any of the above symbols, the API invocation will fail with an exception.
2. Avoiding the above symbols is not enough to guarantee that a Run Name (or a Solution Name) will be a valid file name, because the definition of a valid file name differs between file systems. The policy of the Server API to forbid such symbols should be regarded as a best-effort check rather than an absolute protection. As a result it is still possible to start a configuration instance whose Run Name (or Solution Name) is not a valid file name. Such instances will run into file system related problems if they rely on features like the System Log.

Another consequence is that Solution Names and Run Names must appear in the User Registry instead of absolute file-system paths, for configuration instances which use such names.

Suppose that you have a configuration file with absolute file name "C:/IBM/TDI/configs/rs.xml". The table below describes how to refer to configuration instances launched from that file in the User Registry; the table takes into account whether the configuration file has a Solution Name and whether the configuration instance is started with a Run Name:

Table 75.

Solution Name	Run Name	Section in the User Registry
-	-	[CONFIG]:C:/IBM/TDI/configs/rs.xml
-	myrunname	[CONFIG]:myrunname
mysoluname	-	[CONFIG]:mysoluname
mysoluname	myrunname	[CONFIG]:myrunname

It is important to note that permissions in the User Registry are assigned per configuration instance and not per configuration file.

Using Solution Name instead of Config file path:

Note: Only the configuration files placed in this folder can be edited using the Server API.

In TDI 6.1 and previous releases starting a config instance as well as the check-in/check-out functionality of the Server API required the URL (file path) of the config file to be provided. This is no longer necessary in Tivoli Directory Integrator 7.1, because the same Server API interface methods can be passed the corresponding Solution Name instead. This is a user convenience as Server API clients like the AMC and CLI now accept user-friendly Solution Names instead of cryptic config file paths.

The config file path has a higher priority than the Solution Name. This means that if the method for starting a config instance (for example) is passed a string (either a config file path or the corresponding Solution Name) and it is a valid config file path then the method treats this value as referring to this config file. If there is a config file and a Solution Name which are identical as strings then the config file path takes precedence. This behavior ensures backward compatibility with previous versions of Tivoli Directory Integrator when there were no Solution Names.

At Tivoli Directory Integrator Server startup time, only Configs residing in the Tivoli Directory Integrator configs folder (as specified by the `global.properties` or `solution.properties` file parameter **api.config.folder**) as well as those residing in the Solution Directory can be referred to by their Solution Name.

Scanning the configs folder for Solution Names: At startup the Tivoli Directory Integrator Server scans the configs folder (specified by the **api.config.folder** property in `global.properties` or `solution.properties`) for the Solution Names of the config files located in the configs folder. The Server then builds an internal map which maps Solution Names to config file paths so that Solution Names can be used in place of config file paths.

The following rules are used when scanning the configs folder:

1. If the file name has an extension of ".cfg" – return the file name (these would be very old-style Configs)
2. If a config can be loaded successfully by the config driver, then check for solution name
3. If a config can **not** be loaded by the config driver,
   a. if the file name has an extension of ".xml" – return the file name
   b. different extension – ignore the file and do not return anything

This would lead to the following situations depending on how the Tivoli Directory Integrator server is started:

Table 76.

Tivoli Directory Integrator server in Secure mode	Tivoli Directory Integrator server in Normal mode
PKI-encrypted config – solution name displayed (if existing)	PKI-encrypted config – file name displayed (if extension is .cfg or .xml)
unencrypted config – file name displayed (if extension is .cfg or .xml)	Unencrypted config –Tivoli Directory Integrator solution name displayed (if existing)
password-encrypted config – file name displayed (if extension is .cfg or .xml)	Password-encrypted config – file name displayed (if extension is .cfg or .xml)
non-TDI config file (other, text or binary) – file name displayed (if extension is .cfg or .xml)	Non-Tivoli Directory Integrator config file (other, text or binary) – file name displayed (if extension is .cfg or .xml)

We do not recommend that you store files other than valid Config files (XML format or cfg file format) in the configs folder. During attempts to parse any non-config file, errors may be reported and the file is ignored – this does not affect the proper operation of the Server.

Working with AssemblyLines

Getting access to the AssemblyLines available in a configuration

Assuming that you already have a reference to the Config Instance object, you must obtain the MetamergeConfig object representing the configuration data structure for the whole Config Instance and then get the available AssemblyLines:

```
import com.ibm.di.config.interfaces.MetamergeConfig;
import com.ibm.di.config.interfaces.MetamergeFolder;
import com.ibm.di.config.interfaces.AssemblyLineConfig;

...

MetamergeConfig configuration = configInstance.getConfiguration();
MetamergeFolder configFolder =
    configuration.getDefaultFolder(MetamergeConfig.ASSEMBLYLINE_FOLDER);
String[] assemblyLineNames = configFolder.getNames();
if (assemblyLineNames != null) {
    for (int i=0; i<assemblyLineNames.length; i++) {
        System.out.println(assemblyLineNames[i]);
```

```
// get the AssemblyLine configuration object
AssemblyLineConfig alConfig =
    configuration.getAssemblyLine(assemblyLineNames[i]);
// do something with alConfig ...
```

This block of code prints to the standard output the names of all AssemblyLines in the configuration and demonstrates how to get the AssemblyLine configuration objects. You can use the AssemblyLine configuration object to get more detailed information, such as which Connectors are configured in the AssemblyLine, their parameters, etc.

Note that the MetamergeConfig, MetamergeFolder and AssemblyLineConfig interfaces are not part of the Server API interfaces. They are part of the TDI configuration driver (see the import clauses in the example) and they are not remote objects. When configInstance.getConfiguration() is executed the MetamergeConfig object is serialized and transferred over the wire. Your code will then work with the local copy of that object.

Getting access to running AssemblyLines

You can get the active AssemblyLines either for a specific Config Instance or for all active AssemblyLines on the TDI Server for all running Config Instances.

Getting the active AssemblyLines for a specific Config Instance:

You will need a reference to the Config Instance object. The following code will return all AssemblyLines currently running in the Config Instance:

```
AssemblyLine[] assemblyLines = configInstance.getAssemblyLines();
for (int i=0; i
for (int i=0; i<assemblyLines.length; i++) {
    System.out.println(assemblyLines[i].getName());

    // do someting with assemblyLines[i]
}
```

Getting the active AssemblyLines for the whole TDI Server:

If you want to get all AssemblyLines running on the Server, execute the following code:

```
AssemblyLine[] assemblyLines = session.getAssemblyLines();
for (int i=0; i<assemblyLines.length; i++) {
    System.out.println(assemblyLines[i].getName());

    // do someting with assemblyLines[i]

    // which Config Instance this AssemblyLine belongs to?
    ConfigInstance alConfigInstance = assemblyLines[i].getConfigInstance();
}
```

Note that this is executed at the session level and not for a particular Config Instance. If you need to know which Config Instance a running AssemblyLine belongs to, you can get a reference to the parent Config Instance object through the AssemblyLine object.

You can use the AssemblyLine Server API object to get various AssemblyLine properties, the AssemblyLine configuration object, AssemblyLine log, AssemblyLine result Entry as well as stop the AssemblyLine.

Starting an AssemblyLine

You can start an AssemblyLine through the Config Instance object to which the AssemblyLine belongs. You need to know the name of the AssemblyLine you want to start:

```
AssemblyLine assemblyLine = configInstance.startAssemblyLine("MyAssemblyLine");
```

You also receive a reference to the newly started AssemblyLine instance.

Starting an AssemblyLine in manual mode

The Server API provides a mechanism for manually running an AssemblyLine. In manual mode the AssemblyLine is not running in its own thread. Instead, when you start the AssemblyLine, it is only initialized. Iterations on the AssemblyLine are done in a synchronous manner when the executeCycle() method of the AssemblyLine object is called. This call blocks the current thread and when the AssemblyLine iteration is done it returns the result Entry object.

The following code will start the TestAL AssemblyLine in manual mode and execute three iterations on it. The result Entry from each iteration is printed to the standard output:

```
AssemblyLineHandler alHandler = configInstance.startAssemblyLineManual("TestAL", null);
Entry entry = null;
for (int i=0; i<3; i++) {
  entry = alh.executeCycle();
  System.out.println("TestAL entry: " + entry);
}
alHandler.close();
```

The `startAssemblyLineManual(String aAssemblyLineName, Entry aInputData)` method of the Config Instance object starts an AssemblyLine in manual mode and returns an object of type `com.ibm.di.api.remote.AssemblyLineHandler`. Through this object you can manually iterate through the AssemblyLine, you can pass an initial work Entry and various Task Call Block parameters, you can get a reference to the AssemblyLine Server API object and you can terminate the AssemblyLine when you are done with it.

You can imitate the AssemblyLine runtime behavior by calling `executeCycle()` until it returns `NULL`.

Starting an AssemblyLine with a listener

When you start an AssemblyLine through the Server API you can register a specific AssemblyLine listener that will receive notifications on each AssemblyLine iteration, delivering the result Entry, and also when the AssemblyLine terminates. Through this mechanism you can start an AssemblyLine from a remote application and easily receive all Entries produced by the AssemblyLine. The AssemblyLine listener will also deliver all messages logged during the execution of the AssemblyLine.

Your listener class must implement the `com.ibm.di.api.remote.AssemblyLineListener` interface (or `com.ibm.di.api.local.AssemblyLineListener` for local access).

The methods you must specify are:

- `assemblyLineCycleDone(Entry aEntry)` – this method will be called at the end of each AssemblyLine iteration; the aEntry parameter represents the result Entry from the AssemblyLine iteration.
- `assemblyLineFinished()` – this method is called by the Server API when the AssemblyLine terminates.
- `messageLogged(String aMessage)` – this method is called by the Server API whenever a message is logged through the AssemblyLine logger. Thus you can get remote real time access to the log messages produced by the AssemblyLine.

A sample AssemblyLine listener class that only prints to the standard output all Entries received and all AssemblyLine log messages might look like this:

```
import com.ibm.di.api.DIException;
import com.ibm.di.api.remote.AssemblyLineListener;
import com.ibm.di.entry.Entry;
import java.rmi.RemoteException;

public class MyRemoteALListener implements AssemblyLineListener {

  public void assemblyLineCycleDone(Entry aEntry)
    throws DIException, RemoteException
  {
    System.out.println("AssemblyLine iteration: " + aEntry.toString());
    System.out.println();
```

```
  }

  public void assemblyLineFinished()
    throws DIException, RemoteException
  {
    System.out.println("AssemblyLine terminated.");
    System.out.println();
  }

  public void messageLogged(String aMessage)
    throws DIException, RemoteException
  {
    System.out.println("AssemblyLine log message: " + aMessage);
    System.out.println();
  }
}
```

Once you have implemented your AssemblyLine listener class, you need to instantiate a listener object and pass it when starting the AssemblyLine:

```
MyRemoteALListener alListener = new MyRemoteALListener();
configInstance.startAssemblyLine("TestAL", null,
 AssemblyLineListenerBase.createInstance(alListener,true), true);
```

The startAssemblyLine(String aAssemblyLineName, Entry aInputData, AssemblyLineListener aListener, boolean aGetLogs) method specifies the name of the AssemblyLine, an initial work Entry, the listener object and whether you want to receive log messages – when aGetLogs is false, the messageLogged(String aMessage) listener method will not be called by the Server API.

When you are registering a listener in a remote context, you have to wrap your specific listener in an AssemblyLine Base Listener class – this is necessary to provide a bridge between your custom listener Java class that is not available on the Server side and the Server API notification mechanism. A base listener class is created by calling the static createInstance(AssemblyLineListener aListener, boolean aSSLon) method of the com.ibm.di.api.remote.impl.AssemblyLineListenerBase class. You need to provide the object representing your listener class and specify whether SSL is used for communication with the Server or not (you must specify how the Server API is configured on the Server side – otherwise the communication for that listener will fail).

Starting an AssemblyLine with component simulation

By setting the simulation flag of an AssemblyLine to true you specify that the components behavior in the AssemblyLine will be simulated. The simulation functionality is described in more detail in *IBM Tivoli Directory Integrator V7.1 Users Guide*; here we will only show how to set the simulation flag:

```
usertcb.setProperty(com.ibm.di.server.AssemblyLine.TCB_SIMULATE_MODE, Boolean.TRUE);
```

Where "usertcb" is a TaskCallBlock object, and then start the AL using this object.

Stopping an AssemblyLine

You need a reference to the AssemblyLine object in order to stop it. You can keep the reference to the AssemblyLine object from when you started the AssemblyLine or you can iterate through all running AssemblyLines and find the one you need. Execute the following line of code to stop the AssemblyLine:

```
assemblyLine.stop();
```

Editing configurations

TDI Configurations folder

A TDI Server property `api.config.folder` is available in the TDI Server configuration file `global.properties` - it specifies a folder on the local disk. The Server API will provide calls for browsing and loading configurations placed in this folder or its subfolders. For example:

```
api.config.folder=configs
```

This means that all configuration files placed in "<*TDI_root*>/configs" and its subfolders are eligible for browsing and loading through the Server API (locally and remotely).

The Server API provides new calls for browsing the files and folders in the folder specified by the `api.config.folder` property.

Load for editing

In TDI 6.0 configurations can be edited only after the corresponding Config Instance has been started on the TDI Server. Then there are API calls for getting the Config object, setting the Config object back (probably modified) and saving the configuration on the disk.

Tivoli Directory Integrator 7.1 will not allow modification of the Config object of an active Config Instance. Server API users will still be able to get the Config object for an active Config Instance, but the following calls for setting the Config object and saving it on the disk will throw an exception when executed on a normal running Config Instance:

- `ConfigInstance.setConfiguration(MetamergeConfig configuration)`
- `ConfigInstance.saveConfiguration()`
- `ConfigInstance.saveConfiguration(boolean aEncrypt)`

When a configuration is loaded for editing with a temporary Config Instance it will be able to execute the `setConfiguration(...)` method in order to test the changes applied to the configuration. The `saveConfiguration(...)` methods will however still throw exceptions. Tivoli Directory Integrator 7.1 will present new Server API calls for loading configurations for editing and for saving the edited configurations on the disk.

Configuration Locking

The Server API internally tracks all configurations loaded for editing. When another Server API user requests a configuration already loaded for editing, the method call will fail with exception. A new Server API call has been added for checking whether a configuration is currently loaded for editing (locked).

The lock on a configuration will be released when the user that loaded the configuration for editing saves it back or cancels the update. The Server API provides an option to specify a timeout value for keeping a configuration locked. When that timeout is reached for a configuration the lock is released and the user that locked the configuration will not be able to save it before loading it again.

A new property "api.config.lock.timeout" has been added in the TDI Server configuration file global.properties. It specifies the timeout value in minutes. When the property is left empty or is set a value of 0, this means that there is no timeout. The default value for this property is 0. The timeout logic is implemented by a new thread in the TDI Server. This thread is activated only when "api.config.lock.timeout" is set to a value greater than 0 and will check for and release expired locks each 30 seconds.

A special call for a forced releasing of the lock on a configuration loaded for editing has been added to the Server API. Only Server API users with the admin role will be able to execute it.

All configurations are identified through the relative file path of the configuration file according the TDI Server configurations folder.

All paths specified as method parameters are relative to the TDI Server configurations folder.

The following new calls will be added to the local and remote Server API Session objects and in the JMX interfaces:

- `public boolean releaseConfigurationLock(String aRelativePath) throws DIException;`

  Administratively releases the lock of the specified configuration. This call can be only executed by users with the admin role.

- `public boolean undoCheckOut(String aRelativePath) throws DIException;`

  Releases the lock on the specified configuration, aborting all changes being done. This call can only be executed from a user that has previously checked out the configuration and if the configuration lock has not timed out.
- `public ArrayList listConfigurations(String aRelativePath) throws DIException;`

  Returns a list of the file names of all configurations in the specified folder. The configurations file paths returned are relative to the TDI Server configurations folder.
- `public ArrayList listFolders(String aRelativePath) throws DIException;`

  Returns a list of the child folders of the specified folder
- `public ArrayList listAllConfigurations() throws DIException;`

  Returns a list of the file names of all configurations in the directory subtree of the TDI Server configurations folder. The configurations file paths returned are relative to the TDI Server configurations folder.
- `public MetamergeConfig checkOutConfiguration(String aRelativePath) throws DIException;`

  Checks out the specified configuration. Returns the MetamergeConfig object representing the configuration and locks that configuration on the Server.
- `public MetamergeConfig checkOutConfiguration(String aRelativePath, String aPassword) throws DIException;`

  Checks out the specified password protected configuration. Returns the MetamergeConfig object representing the configuration and locks that configuration on the Server.
- `public void checkInConfiguration(MetamergeConfig aConfiguration, String aRelativePath) throws DIException;`

  Saves the specified configuration and releases the lock. If a temporary Config Instance has been started on check out, it will be stopped as well.
- `public void checkInConfiguration(MetamergeConfig aConfiguration, String aRelativePath, boolean aEncrypt) throws DIException;`

  Encrypts and saves the specified configuration and releases the lock. If a temporary Config Instance has been started on check out, it will be stopped as well.
- `public void checkInAndLeaveCheckedOut(MetamergeConfig aConfiguration, String aRelativePath) throws DIException;`

  Checks in the specified configuration and leaves it checked out. The timeout for the lock on the configuration is reset.
- `public MetamergeConfig createNewConfiguration(String aRelativePath, boolean aOverwrite) throws DIException;`

  Creates a new empty configuration and immediately checks it out. If a configuration with the specified path already exists and the aOverwrite parameter is set to false the operation will fail and an Exception will be thrown.
- `public ConfigInstance checkOutConfigurationAndLoad(String aRelativePath) throws DIException;`

  Checks out the specified configuration and starts a temporary Config Instance on the Server. This Config Instance will be stopped when the configuration is checked in or when the lock on the configuration expires. The method returns the ConfigInstance object. The MetamergeConfig object can be retrieved through the ConfigInstance object.
- `public ConfigInstance checkOutConfigurationAndLoad(String aRelativePath, String aPassword) throws DIException;`

  Checks out the specified password protected configuration and starts a temporary Config Instance on the Server. This Config Instance will be stopped when the configuration is checked in or when the lock on the configuration expires. The method returns the ConfigInstance object. The MetamergeConfig object can be retrieved through the ConfigInstance object.
- `public ConfigInstance createNewConfigurationAndLoad(String aRelativePath, boolean aOverwrite) throws DIException;`

Creates a new empty configuration, immediately checks it out and loads a temporary Config Instance on the Server. If a configuration with the specified path already exists and the aOverwrite parameter is set to false the operation will fail and an Exception will be thrown. The temporary Config Instance will be stopped when the configuration is checked in or when the lock on the configuration expires. The method returns the ConfigInstance object. The MetamergeConfig object can be retrieved through the ConfigInstance object.

- `public boolean isConfigurationCheckedOut(String aRelativePath) throws DIException;`

  Checks if the specified configuration is checked out on the Server.

Load for editing with temporary Config Instance

This is a special version of the load for edit mechanism – the difference is that when the configuration is loaded for editing a temporary Config Instance will be started as well. This will allow testing the configuration and its AssemblyLines while they are being developed and will be particularly useful for development tools like the TDI Config Editor.

The Config Instance will be automatically stopped when the configuration is released or when the lock on the configuration expires.

The temporary Config Instances are independent of the normal long running Config Instances on the Server. A normal Config Instance from configuration rs.xml might be running on the Server and at the same time the rs.xml configuration can be loaded for editing with a temporary Config Instance. This will result in starting a new temporary Config Instance from the rs.xml file in addition to the normal long running rs.xml Config Instance.

The same locking mechanism applies for configurations loaded for editing with a temporary Config Instance. This means that a configuration can be loaded for editing only once regardless of whether it has been loaded for editing with a temporary Config Instance or without.

Using the Solution Name instead of the config file path

Traditionally, starting a config instance as well as the check-in/check-out functionality of the Server API required the URL (file path) of the config file to be provided. This is no longer necessary in the current version of Tivoli Directory Integrator, because the same Server API interface methods can be passed the corresponding Solution Name instead. This is a user convenience as Server API clients like the AMC and CLI can take user-friendly Solution Names from the user instead of cryptic config file paths.

Config file path precedence over Solution Names:

The config file path has a higher priority than the Solution Name. This means that if the method for starting a config instance is passed a string (either a config file path or the corresponding Solution Name) and it is a valid config file path then the method will treat this value as referring to this config file. This means that if there is a config file and a Solution Name which are identical as strings then the config file path takes precedence. This behavior ensures backward compatibility with previous versions of TDI when there were no Solution Names.

Configs Folder:

Only config files residing in the Tivoli Directory Integrator *configs* folder at Tivoli Directory Integrator Server startup time can be referred to by their Solution Name.

See also "Optional Config instance ID in a Config file" in *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide* for more information about Solution Names, Run Names and how to configure these.

Server API event for configuration update

A Server API event `di.ci.file.updated` will be fired whenever a configuration that has been locked is saved on the TDI Server.

This notification will allow Server API clients to get notified for changes in configurations they are using and for example reload them to get the latest version.

Working with the System Queue

The System Queue is a TDI server module which TDI internal objects as well as TDI components can use as a general purpose queue. The purpose of the System Queue is to connect to a JMS Provider and provide functionality for getting from JMS message queues and putting into JMS message queues general messages as well as TDI Entry objects. The System Queue can connect to different JMS Providers using different TDI JMS Drivers. For more information on the System Queue please see the "System Queue" chapter of the *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide*.

The System Queue functionality is exposed through both the local and remote interfaces of the Server API as well as through the JMX layer of the Server API. TDI components and subsystems which run in the Java Virtual Machine of the local TDI server are expected to use the local Server API interfaces to interact with the System Queue. Remote Server API client applications as well as TDI components and sub-systems which run in the Java Virtual Machine of a remote TDI server are expected to use the remote Server API interfaces.

The Tivoli Directory Integrator 7.1 Server API JMX layer contains a SystemQueue MBean. This MBean provides JMX access to the SystemQueue. A JMX client can access the SystemQueue JMX MBean and thus work with the System Queue through JMX.

The System Queue must be properly configured before it can be accessed through the Server API. A simple way to configure the System Queue is like the following procedure:

- Setup the JMS Provider.

  TDI provides the MQ Everyplace JMS Provider out of the box. You can setup a MQe Queue Manager via the *mqeconfig* command line utility (the *mqeconfig* utility is located in the 'jars/plugins' subfolder of your TDI installation).

  Modify the mqeconfig.props configuration file.
  - Specify the folder where you want to place the MQe Queue Manager:

    ```
    serverRootFolder=C:\\TDI\\MQePWStore
    ```
  - Specify the IP address of the TDI Server:

    ```
    serverIP=127.0.0.1
    ```
  - Having the configuration options set, create the MQe Queue Manager:

    ```
    mqeconfig mqeconfig.props create server
    ```
  - Create a queue for test purposes:

    ```
    mqeconfig mqeconfig.props create queue myqueue
    ```
- Configure the System Queue and the JMS Provider in `global.properties` or `solution.properties`
  - Turn on System Queue usage:

    ```
    systemqueue.on=true
    ```
  - Set the JMS driver for the System Queue to MQ Everyplace:

    ```
    systemqueue.jmsdriver.name=com.ibm.di.systemqueue.driver.IBMMQe
    ```
  - Set the configuration file for the MQe Queue Manager (this file has been generated by the mqeconfig utility):

    ```
    systemqueue.jmsdriver.param.mqe.file.ini=C:\TDI\MQePWStore\pwstore_server.ini
    ```

:

Note: For a stand-alone Java program to operate successfully with the System Queue through the Server API, a JMS implementation must be included in the CLASSPATH of the program. You can use the JMS implementation distributed with Tivoli Directory Integrator: `jars/3rdparty/IBM/ibmjms.jar`

Access the System Queue through the Server API

Once a Server API session is initiated, the System Queue can be accessed like this:

```
import com.ibm.di.api.remote.SystemQueue;
...
SystemQueue systemQueue = session.getSystemQueue();
```

Put a message in the System Queue

The following code puts a text message into a queue named "myqueue" (the call will not create the specified queue automatically - the queue must be created manually first):

```
systemQueue.putTextMessage("myqueue", "mytextmessage");
```

Retrieve a message from the System Queue

The following code retrieves a text message from a queue named "myqueue" (the queue must exist). The method call waits a maximum of 10 seconds for a message to become available:

```
String textMessage = systemQueue.getTextMessage("myqueue", 10);
```

Working with the Tombstone Manager

Previous versions of TDI do not keep track of configurations or AssemblyLines that have terminated. Therefore, administrators have no way of knowing when their AssemblyLines last ran, without going into the log of each one. Bundlers that initiate AssemblyLines have no way of querying their status after they've terminated.

The solution is a Tombstone Manager that creates records ("tombstones") for each AssemblyLine and configuration as they terminate, that contain exit status and other information that later can be requested through the Server API.

Globally Unique Identifiers

Globally Unique Identifiers (GUID) are created by the Server API to uniquely identify Config Instance and AssemblyLine instances. The GUID is a string value that is unique for each instance of a Config Instance, an AssemblyLine or an EventHandler (from older versions of Tivoli Directory Integrator) ever created by a particular Tivoli Directory Integrator Server.

GUIDs are defined as the string representation of the Config Instance/AssemblyLine object hashcode concatenated with the string representation of the Config Instance/AssemblyLine start time in milliseconds.

A method is available in the Config Instance and AssemblyLine Server API interfaces: `String getGlobalUniqueID ();`

A field GlobalUniqueID is available in the AssemblyLine and Config Instance stop Server API events.

Server API support for the Tombstone Manager

What is a tombstone: The Server API provides a new class `com.ibm.di.api.Tombstone` whose instances represent tombstone objects. The public interface of the Tombstone class follows:

```
public class Tombstone implements Serializable {

    public int getComponentTypeID ()

    public int getEventTypeID ()

    public java.utilDate getStartTime ()

    public java.utilDate getTombstoneCreateTime ()

    public String getComponentName ()

    public String getConfigID ()
```

```
    public int getExitCode ()

    public String getErrorDescription ()

    public String getGUID ()

    public Entry getStat ()

    public String getUserMessage ()

}
```

Retrieving tombstones: Tombstones are retrieved through the Tombstone Manager. You can access the Tombstone Manager via the Server API like this:

```
import com.ibm.di.api.remote.TombstoneManager;
...
TombstoneManager tombstoneManager = session.getTombstoneManager();
```

With the Tombstone Manager at hand, you can search for specific Tombstones. The following code iterates through all tombstones created last week:

```
Calendar calendar = Calendar.getInstance();
calendar.add(Calendar.DATE, -7);

Tombstone[] tombstones = tombstoneManager.getTombstones(calendar.getTime(), new Date());

for (int i = 0; i < tombstones.length; ++i) {
System.out.println("Tombstone found for : "+tombstones[i].getComponentName());
System.out.println("\t GUID : "+tombstones[i].getGUID());
System.out.println("\t statistics : "+tombstones[i].getStatistics());
}
```

All tombstones for a particular AssemblyLine can be retrieved this way (the example AssemblyLine is named "myline" and the ID of the configuration is "C__TDI_myconfig.xml"):

```
Tombstone[] alTombstones = tombstoneManager.getAssemblyLineTombstones("AssemblyLines/myline",
 "C__TDI_myconfig.xml");
```

The following new Server API calls are provided for querying the Tombstone Manager – these are methods of the `com.ibm.di.api.local.TombstoneManager` interface:

- `Tombstone getTombstone (String aGUID)`

  Returns a single tombstone object uniquely identified by the specified GUID.
- `Tombstone[] getAssemblyLineTombstones (String aAssemblyLineName, String aConfigID)`

  Returns all available tombstones for the specified AssemblyLine.
- `Tombstone[] getAssemblyLineTombstones (String aAssemblyLineName, String aConfigID, java.util.Date aStartTime, java.util.Date aEndTime)`

  Returns all available tombstones for the specified AssemblyLine with timestamps in the interval specified by aStartTime and aEndTime.
- `Tombstone[] getConfigInstanceTombstones (String aConfigID)`

  Returns all available tombstones for the specified Config Instance.
- `Tombstone[] getConfigInstanceTombstones (String aConfigID)`

  Returns all available tombstones for the specified Config Instance.
- `Tombstone[] getTombstones (java.util.Date aStartTime, java.util.Date aEndTime)`

  Returns all available tombstones with timestamps in the interval specified by aStartTime and aEndTime.

Deleting tombstones: When tombstones are no longer needed they should be deleted.

The following code deletes all tombstones from the last week:

```
tombstoneManager.deleteTombstones(7);
```

The following Server API calls are provided for deleting old tombstone records:

- `int deleteTombstones (int aDays)`

  Deletes all tombstones that are older than the specified number of days. Returns the number of deleted tombstone records.
- `int keepMostRecentTombstones (int aMostResentToKeep)`

  After this method is executed only the *aMostRecentToKeep* most recent tombstone records are kept and all other are deleted. Returns the number of deleted tombstone records.
- `int deleteALTombstones (String aAssemblyLineName, String aConfigID)`

  Deletes all tombstones for specified AssemblyLine. Returns the number of deleted tombstone records.
- `int deleteALTombstones (String aAssemblyLineName, String aConfigID, int aDays)`

  Deletes all tombstones for the specified AssemblyLine that are older than the specified number of days. Returns the number of deleted tombstone records.
- `int keepMostRecentALTombstones (String aAssemblyLineName, String aConfigID, int aMostResentToKeep)`

  After this method is executed only the *aMostRecentToKeep* most recent tombstone records for the specified AssemblyLine are kept and all other are deleted. Returns the number of deleted tombstone records.
- `int deleteCITombstones (String aConfigID)`

  Deletes all tombstones for specified Config Instance. Returns the number of deleted tombstone records.
- `int deleteCITombstones (String aConfigID, int aDays)`

  Deletes all tombstones for the specified Config Instance that are older than the specified number of days. Returns the number of deleted tombstone records.
- `int keepMostRecentCITombstones (String aConfigID, int aMostResentToKeep)`

  After this method is executed only the *aMostRecentToKeep* most recent tombstone records for the specified Config Instance are kept and all other are deleted. Returns the number of deleted tombstone records.
- `boolean deleteTombstone (String aGUID)`

  Deletes the tombstone with the specified GUID. Returns true only when the tombstone object with the specified GUID is found and deleted.

Adding a custom message to AssemblyLine tombstones

The *task* script object represents the AssemblyLine object in an AssemblyLine context so that you can use this object when scripting.

The interface of the *task* object is extended to provide a method for setting a custom message that will be saved in the UserMessage field of the tombstone for this AssemblyLine. The signature of the new method, accessible through the task script object is as follows:

```
task.setTombstoneUserMessage(String aUserMessage);
```

This method can be used from AssemblyLine scripts to provide additional information in the AssemblyLine tombstone.

The user message of a tombstone can be retrieved like this:

```
String userMessage = tombstone.getUserMessage();
```

Note: No user defined messages can be set for ConfigInstance tombstones.

Working with TDI Properties

For a remote client to query/get/set properties (or stores), it needs to be provided a remote reference of the TDIProperties object in the server. A remote client can obtain the com.ibm.di.api.remote.TDIProperties interface remote reference via the following method in com.ibm.di.api.remote.ConfigInstance:

```
public TDIProperties getTDIProperties() throws DIException,RemoteException;
```

A similar interface and implementation is available in the local Server API interfaces.

For a description of the interface methods please see the TDI JavaDocs.

The following example lists all available Property Stores for a given configuration instance:

```
TDIProperties tdiProperties = configInstance.getTDIProperties();

List stores = tdiProperties.getPropertyStoreNames();
Iterator it = stores.iterator();
System.out.println("Available property stores :");
while (it.hasNext()) {
 String storeName = (String) it.next();
 System.out.println("\t"+storeName);
}
```

Individual properties can be acquired by their name. The following code prints all properties available in the Global Property Store (global.properties) :

```
String storeName = "Global-Properties";
System.out.println(storeName+" store contents :");
String[] storeKeys = tdiProperties.getPropertyStoreKeys(storeName);
for (int i = 0; i < storeKeys.length; ++i) {
System.out.println("\t"+storeKeys[i]+" : "+ tdiProperties.getProperty(storeName, storeKeys[i]));
}
```

Property values can be changed and new properties can be created like this:

```
tdiProperties.setProperty(storeName, "mykey", "myvalue");
```

The following code removes a property from a Property Store:

```
tdiProperties.removeProperty(storeName, "mykey");
```

Before any changes to a Property Store (adding a new property, changing the value of a property or removing a property) take effect, the changes must be committed:

```
tdiProperties.commit();
```

JMX layer API

A TDIPropertiesMBean interface is available in the com.ibm.di.api.jmx.mbeans package. The methods exposed in TDIPropertiesMBean interface are similar to the ones exposed in the com.ibm.di.api.remote.TDIProperties interface.

A method getTDIProperties( ) is available in the com.ibm.di.api.jmx.mbeans.ConfigInstanceMBean class via which a JMX client can obtain a reference to a javax.management.ObjectName interface.

Registering for Server API event notifications

The Server API provides an event notification mechanism for Server events like starting and stopping of Config Instances and AssemblyLines. This allows a local or remote client application to register for event notifications and react to various events.

Applications that need to register and receive notifications should implement a listener class that implements the DIEventListener interface (com.ibm.di.api.remote.DIEventListener for remote

applications and com.ibm.di.api.local.DIEventListener for local access). This class is responsible for processing the Server events. The handleEvent(DIEvent aEvent) method from the DIEventListener interface is where you need to put your code that processes Server events. Of course you may implement as many listener classes as you need, with different implementations of the handleEvent(DIEvent aEvent) method and register all of them as event listeners. A sample listener that just logs the event object might look like this:

```
import java.rmi.RemoteException;

import com.ibm.di.api.DIEvent;
import com.ibm.di.api.DIException;
import com.ibm.di.api.remote.DIEventListener;

public class MyListener implements DIEventListener
{
  public void handleEvent (DIEvent aEvent) throws DIException, RemoteException
  {
    System.out.println("TDI Server event: " + aEvent);
    System.out.println();
  }
}
```

Once you have implemented your listener you will need to register it with the Server API. If however you are implementing a remote application there is one extra step you need to perform before actually registering the listener object with the Server API – you need to instantiate and use a base listener object that will wrap the listener you implemented. The base listener class allows you to use your own listener classes without having the same Java classes available on the Server:

```
DIEventListener myListener = new MyListener();
DIEventListener myBaseListener = DIEventListenerBase.createInstance(myListener, true);
```

The base listener object implements the same *DIEventListener* interface – its class however is already present on the Server and it can act as a bridge between your local client side listener class and the Server. A base listener object is created by calling the static method *createInstance(DIEventListener aListener, boolean aSSLon)* of the *com.ibm.di.api.remote.impl.DIEventListenerBase* class. The first parameter *aListener* represents the actual listener object and the second one specifies whether SSL is used or not by the Server API (note that this is not an option for you to select whether to use SSL or not with this listener object; here you have to specify how the Server API is configured on the Server side – otherwise the communication for that listener will fail).

When you have your listener object ready (or a base listener for remote access), you can register for event notifications through the session object:

```
session.addEventListener(myBaseListnener, "di.*", "*");
```

The *addEventListener(DIEventListener aListener, String aTypeFilter, String aIdFilter)* method of the session object will register your listener. The first parameter *aListener* is the listener object (or the base listener object for remote access), *aTypeFilter* and *aIdFilter* let you specify what types of events you want to receive:

- *aTypeFilter* specifies what type of event objects you want to receive. The currently supported events are:
  - **di.ci.start** – Config Instance started
  - **di.ci.stop** – Config Instance stopped
  - **di.al.start** – AssemblyLine started
  - **di.al.stop** – AssemblyLine stopped
  - **di.ci.file.updated** – Configuration file modified
  - **di.server.stop** – TDI Server shutdown

  You can either specify a specific event type like di.al.start or you can specify a filter using the "*" wildcard; for example di.al.* will register your listener for all Server events related to AssemblyLines, while a type filter of * or NULL will register your listener for all events.

- *aIdFilter* is only taken into account when *aTypeFilter* is not set to "*" or NULL. It lets you filter events depending on the object related to the event – for AssemblyLines this is the AssemblyLine name, and for Config Instances this is the Config Instance ID. For example, if you register your listener with `addEventListener(myListnener, "di.al.start", "MyAssemblyLine")` it will only be sent events when the "MyAssemblyLine" AssemblyLine is started and will not receive any other Server events.

If at some point you want to stop receiving event notifications from a listener already registered with the Server API, you need to unregister the listener. This is done through the same session object it was registered with by calling:

```
session.removeEventListener(myListener);
```

Server shutdown event

A new Server API event notification has been added to signal Server shutdown events. This event is available to Server API clients and JMX clients, both in local and remote context. The event type is "di.server.stop" for both the Server API and JMX notification layers. As an additional user data the event object conveys the Server boot time.

Custom Server API event notifications

New Server API functionality has been added for sending custom, user defined event notifications. The following new call has been added to the local and remote Server API Session objects and also to the *DIServer* MBean so that it can be accessed from the JMX context as well:

```
public void sendCustomNotification (String aType, String aId, Object aData)
```

The invocation of this method will result in broadcasting a new user defined event notification. The parameters that must be passed to this method have the same meaning as the respective parameters of standard Server API notifications. The aType parameter specifies the type of the event. The value given by the user will be prefixed with the `user.` prefix. For example if the type passed by the user is `process.X.completed` the type of the event broadcast will be `user.process.X.completed`. A client application can register for all custom events specifying a type filter of `user.*`. The aId parameter can be used to identify the object this event originated from. The standard Server API events use this value to specify a Config Instance or AssemblyLine. The aData parameter is where the user can pass on any additional data related to this event; if the event is expected to be sent and received in a remote context, this object has to be serializable.

Getting access to log files

"Starting an AssemblyLine with a listener" on page 546 describes how listeners can be used to get AssemblyLine log messages in real time as they are produced.

The Server API provides another mechanism for direct access to log files produced by AssemblyLines. This mechanism only provides access to the log files generated by the AssemblyLine SystemLog logger.

You don't need a reference to an AssemblyLine Server API object to get to the log file. Also you can access old logs of AssemblyLines that have terminated.

First you need to get hold of the `SystemLog` object:

```
SystemLog systemLog = session.getSystemLog();
```

You can then ask for all the log files generated by an AssemblyLine:

```
String[] alLogFileNames = systemLog.getALLogFileNames("C__Dev_TDI_rs.xml", "TestAL");
if (alLogFileNames != null) {
  System.out.println("Availalbe AssemblyLine log files:");
  for (int i=0; i<allLogFileNames.length; i++) {
    System.out.println(alLogFileNames[i]);
  }
}
```

The *getALLogFileNames(String aConfigId, String aALName)* method is passed the Config ID (see "Stopping a Config Instance" on page 541 for more details on the Config ID) and the name of the AssemblyLine. This will return an array with the names of all log files generated by runs of the specified AssemblyLine.

If you are interested in the last run of the AssemblyLine only, there is a Server API call that will give you the name of that log file only:

```
String lastALLogFileName = systemLog.getALLastLogFileName("C__Dev_TDI_rs.xml", "TestAL");
System.out.println("AssemblyLine last log file name: " + lastALLogFileName);
```

When you have got the name of a log file you can retrieve the actual content of the log file:

```
String alLog = systemLog.getALLog("C__Dev_TDI_rs.xml", "TestAL", lastALLogFileName);
System.out.println("TestAL AssemblyLine log: ");
System.out.println(alLog);
```

In cases where the log file can be huge, you might want to retrieve only the last chunk of the log. The sample code below specifies that only the last 10 kilobytes from the log file should be retrieved:

```
String alLog = systemLog.getALLogLastChunk("C__Dev_TDI_rs.xml", "TestAL", lastALLogFileName, 10);
System.out.println("Last 10K of the TestAL AssemblyLine log: ");
System.out.println(alLog);
```

The Server API also provides methods for cleaning up (deleting) old log files.

You can delete all log files (for all configurations and all AssemblyLines) older than a specific date. The sample code below will delete all log files older than a week:

```
Calendar calendar = Calendar.getInstance();
calendar.add(Calendar.DATE, -7);
systemLog.cleanAllOldLogs(calendar.getTime());
```

Another criterion you can use for log files clean up is the number of log files for each AssemblyLine. You can specify that you want to delete all log files except the 5 most recent logs for all AssemblyLines:

```
systemLog.cleanAllOldLogs(5);
```

You can also delete the log files for AssemblyLines only or for a specific AssemblyLine. The same two criteria are available: date and number of log files but in addition you can specify the name of an AssemblyLine or use calls that operate on all AssemblyLines. Consult the JavaDoc of the `com.ibm.di.api.remote.SystemLog` or `com.ibm.di.api.local.SystemLog` interfaces for the signatures and the descriptions of all log clean up methods.

Server Info

Through the Server API you can get various types of information about the TDI Server itself like the Server version, IP address, operating system, boot time and information about what Connectors, Parsers and Function Components are installed and available on the Server.

It is the `ServerInfo` object that provides access to this information. You can get the `ServerInfo` object through the session object:

```
ServerInfo serverInfo = session.getServerInfo();
```

You can then get and print out details of the Server environment:

```
System.out.println("Server IP address: " + serverInfo.getIPAddress());
System.out.println("Server host name: " + serverInfo.getHostName());
System.out.println("Server boot time: " + serverInfo.getServerBootTime());
System.out.println("Server version: " + serverInfo.getServerVersion());
System.out.println("Server operating system: " + serverInfo.getOperatingSystem());
```

You can also output a list of all Connectors installed and available on the Server:

```
String[] connectorNames = serverInfo.getInstalledConnectorsNames();
System.out.println("Connectors available on the Server: ");
for (int i=0; i<connectorNames.length; i++) {
  System.out.println(connectorNames[i]);
}
```

You can output more details for each installed Connector including its description and version:

```
String[] connectorNames = serverInfo.getInstalledConnectorsNames();
for (int i=0; i<connectorNames.length; i++) {
  System.out.println("Installed connector: ");
  System.out.println("    name: " + connectorNames[i]);
  System.out.println("    description: " + serverInfo.getConnectorDescription(connectorNames[i]));
  System.out.println("    version: " + serverInfo.getConnectorVersionInfo(connectorNames[i]));
  System.out.println();
}
```

In information for other components can be retrieved in a similar manner – Parsers and Functional Components.

Using the Security Registry

The Security Registry is a special Server API object that lets you query what rights a user is granted and whether he/she is authorized to execute a specific action. This is useful if an application is building an authentication and authorization logic of its own – for example the application is using internally a single admin user for communication with the TDI Server and it manages its own set of users and rights.

The Security Registry object is only available to users with the admin role. It is obtained through the session object:

```
SecurityRegistry securityRegistry = session.getSecurityRegistry();
```

You can then check various user rights. For example, `securityRegistry.userIsAdmin("Stan")` will return true if Stan is granted the admin role; `securityRegistry.userCanExecuteAL ("User1", "rs.xml", "TestAL")` will return true only if Stan is allowed to execute AssemblyLine "TestAL" from configuration "rs.xml".

Check the JavaDoc of `com.ibm.di.api.remote.SecurityRegistry` for all available methods.

Custom Method Invocation

You sometimes need to implement your own functionality and be able to access it from the Server API, both locally and remotely. This was supported by the Server API in TDI 6.0, but it needed to be simplified so that you can drop a JAR file of your own in the TDI classpath and then access it from the Remote Server API without having to deal with RMI.

Two methods are now available in the following interfaces:

- `com.ibm.di.api.remote.Session`
- `com.ibm.di.api.local.Session`

The two methods are:

```
public Object invokeCustom(String aCustomClassName, String aMethodName, Object[] aParams)
 throws DIException;
```

and

```
public Object invokeCustom(String aCustomClassName, String aMethodName,
 Object[] aParamsValue, String[] aParamsClass)
 throws DIException;
```

Both methods invoke a custom method described by its class name, method name and method parameters.

These methods can invoke only static methods of the custom class. This is not a limitation, because the static method of the custom class can instantiate an object of the custom class and then call instance methods of the custom class.

The main difference between the two methods is that the `invokeCustom(String, String, Object[], String[])` method requires the type of the parameters to be explicitly set (in the `paramsClass` String array) when invoking the method. This helps when you want to invoke a custom method from a custom class, but also want to invoke this method with a null parameter value. Since the parameter's value is null its type can not be determined and so the desired method to be called cannot be determined.

If the you need to invoke a custom method with a null value you must use the `invokeCustom(String, String, Object[], String[])` method, where the desired method is determined by the elements of the String array which represents the types and the exact order of the method parameters. If the user uses `invokeCustom(String, String, Object[])` and in the object array put a value which is null than an Exception will be thrown.

Primitive types handling
: These methods do not support the invocation of a method with primitive types of parameter(s). All primitive types in Java have a wrapper class which could be used instead of the primitive type.

Custom methods with no parameters
: If your need to invoke a method which has no parameters you must set the `paramsValue` object array to null (and the `paramsClass` String array if the other method is used).

Errors
: Several exceptions may occur when using these methods. Both local and remote sessions support these two methods, but the Server API JMX does not.

Turning custom invocation on/off
: The ability to use `invokeCustom()` methods can be turned on or off (the default is off). This can be done by setting a property in the global.properties file named `api.custom.method.invoke.on` to `true` or `false`. If the value of this property is set to true then users can use these methods.

Specifying the classes allowed for custom invocation
: There is a restriction on the classes which can be invoked by these Server API methods. In the global.properties file there is another property named `api.custom.method.invoke.allowed.classes` which specifies the list of classes which these methods can invoke. If these methods are used and a class which is not in the list of allowed classes is invoked then an exception is thrown. The value of this property is the list of fully qualified class names separated by comma, semicolon, or space.

Examples
: Here are some sample values for these properties:

```
api.custom.method.invoke.on=true
api.custom.method.invoke.allowed.classes=com.ibm.MyClass,com.ibm.MyOtherClass
```

The first line of this example specifies that custom invocation is turned on and thus the two `invokeCustom()` methods are allowed to be used. The second line specifies which classes can be invoked. In this case only `com.ibm.MyClass` and `com.ibm.MyOtherClass` classes are allowed to be invoked. If one of the two `invokeCustom()` methods is used to invoke a different class then an exception is thrown.

Defaults
: The default value of the `api.custom.method.invoke.on` property is `false`. This means that users are not allowed to use the two `invokeCustom()` methods and that an exception would be thrown

if any one of these methods is invoked. The default value of the `api.custom.method.invoke.allowed.classes` is empty, in other words, no classes can be invoked. This means that even if custom invocation is turned on no classes can be invoked by the two `invokeCustom()` methods.

A Full Example

Suppose the following class is packaged in a jar file, which is then placed in the 'jars' folder of TDI:

```
public class MyClass {
 public static Integer multiply(Integer a, Integer b) {
  return new Integer(a.intValue() * b.intValue());
 }
}
```

Suppose the global.properties TDI configuration file contains the following lines:

```
api.custom.method.invoke.on=true
api.custom.method.invoke.allowed.classes=MyClass
```

Then in a client application the 'multiply' method of 'MyClass' can be invoked in a Server API session like this:

```
Integer result = (Integer) session.invokeCustom(
                           "MyClass",
                          "multiply",
           new Object[] {new Integer(3), new Integer (5)});
```

The JMX layer

The Server API provides a JMX layer. It exposes all Server API calls through a JMX interface locally and remotely (through the JMX Remote API 1.0).

Please refer to the "Remote Server" chapter in the *IBM Tivoli Directory Integrator V7.1 Installation and Administrator Guide* for information on how to switch on and setup the JMX layer of the Server API for local and remote access.

Local access to the JMX layer

You can get a reference to the `JMX MBeanServer` object from the local Server JVM by calling

```
import com.ibm.di.api.jmx.JMXAgent;
import javax.management.MBeanServer;

...

MBeanServer jmxMBeanServer = JMXAgent.getMBeanServer();
```

The `getMBeanServer()` static method of the `com.ibm.di.api.jmx.JMXAgent` class will return an `MBeanServer` JMX object that represents an entry point to all MBeans provided by the JMX layer of the Server API. You can also register for JMX notifications with the `MBeanServer` object returned.

Note: The `getMBeanServer()` method will throw an Exception if it is called and the JMX layer of the Server API is not initialized.

Remote access to the JMX layer

The remote JMX access to the Server API is implemented as per the JMX Remote API 1.0 specification.

You have to use the following JMX Service URL for remote access:

```
service:jmx:rmi://<TDI_Server_host>/jndi/rmi://<TDI_Server_host>:<TDI_Server_RMI_port>/jmxconnector
```

You need to replace <TDI_Server_host> and <TDI_Server_RMI_port> with the host and the RMI port of the TDI Server; for example, `service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxconnector`

The sample code below demonstrates how a remote JMX connection can be established:

```
import javax.management.MBeanServerConnection;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

...

JMXServiceURL jmxUrl = new
  JMXServiceURL("service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxconnector");
JMXConnector jmxConnector = JMXConnectorFactory.connect(jmxUrl);
MBeanServerConnection jmxMBeanServer = jmxConnector.getMBeanServerConnection();
```

Similarly to the local JMX access the *MBeanServerConnection* object is the entry point to all MBeans and notifications provided by the JMX layer of the Server API.

For example, you can list all MBeans available on the JMX Server:

```
Iterator mBeans = jmxMBeanServer.queryNames(null, null).iterator();
while (mBeans.hasNext()) {
  System.out.println("MBean: " + mBeans.next());
}
```

MBeans and Server API objects

The JMX layer wraps the Server API objects in MBeans. The access to the MBeans is however straightforward - you can directly look up an MBean through the `MBeanServerConnection` object.

There is no session object in the MBean layer (the session and the security checks are managed through the RMI session). The methods for creating, starting and stopping Config Instances that exist in the Server API Session object can be found in the DIServer MBean in the JMX layer.

A list of the Server API MBeans available at some time on a TDI Server might look like this:

- `ServerAPI:type=ServerInfo,id=192.168.113.222`
- `ServerAPI:type=ConfigInstance,id=C__Dev_TDI_11_11_fp1_rs.xml`
- `ServerAPI:type=AssemblyLine,id=AssemblyLines/longal.618794016`
- `ServerAPI:type=DIServer,id=winserver`
- `ServerAPI:type=SystemLog,id=SystemLog`
- `ServerAPI:type=SecurityRegistry,id=SecurityRegistry`
- ServerAPI:type=Notifier,id=Notifier

Each Config Instance or AssemblyLine is wrapped in an MBean. When the Config Instance or AssemblyLine is started the MBean is created automatically and it is automatically removed when the Config Instance or AssemblyLine terminates.

Refer to the JavaDoc of the Java package `com.ibm.di.api.jmx.mbeans` for all available MBeans, their methods and attributes.

JMX notifications

The JMX layer of the Server API provides local and remote notifications for all Server API events (see "Working with the System Queue" on page 551.)

You have to register for JMX notifications with the Notifier MBean.

The JMX notification types are exactly the same as the Server API notifications:

- di.ci.start – Config Instance started
- di.ci.stop – Config Instance stopped
- di.al.start – AssemblyLine started
- di.al.stop – AssemblyLine stopped
- di.ci.file.updated – Configration file modified
- di.server.stop – TDI Server shutdown

JMX Example - Tivoli Directory Integrator 7.1 and MC4J configuration

This example describes how MC4J and TDI can be set up so that MC4J can be used to access the Server API JMX layer from MC4J.

TDI side

Set up Remote Server API and JMX: Set the following properties in `global.properties` or `solution.properties` file (the long hexadecimal values may run off the side of this document):

```
## Server API properties
## --------------------

api.on=true
api.user.registry=serverapi/registry.txt
api.user.registry.encryption.on=false

api.remote.on=true
api.remote.ssl.on=false
api.remote.ssl.client.auth.on=true
api.remote.naming.port=1099
# api.remote.server.ports=8700-8900
api.truststore=testserver.jks
{protect}-api.truststore.pass={encr}L79kdqak1afKdAyuCZBMi1GqY/DPfD1Ipo020CVAGx/OROE2JBUTgZxLjqADXSZJgM3dHg2aWlCRw

## Specifies a list of IP addresses to accept non SSL connections from (host names are not accepted).
## Use space, comma or semicolon as delimiter between IP addresses. This property is only taken into account
## when api.remote.ssl.on is set to false.
## api.remote.nonssl.hosts=

api.jmx.on=true
api.jmx.remote.on=true
```

Note: SSL is turned off for easy configuration.

Property `api.remote.server.ports` specifies which ports are used for the RMI services; this property can be used to change the default range (8700-8900) if there is a firewall between the server and the client, and the firewall requires this.

Start the TDI server from the command line:

```
D:\TDI>ibmdisrv -d

CTGDKD435I Remote API successfully started on port:1099, bound to:'SessionFactory'. SSL and Client Authentication are disabled.
CTGDKD111I JMX Remote Server Connector started at: service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxconnector.
```

MC4J side

1. Download and install MC4J from http://sourceforge.net/projects/mc4j/.
2. Start the **Connect to server ...** wizard

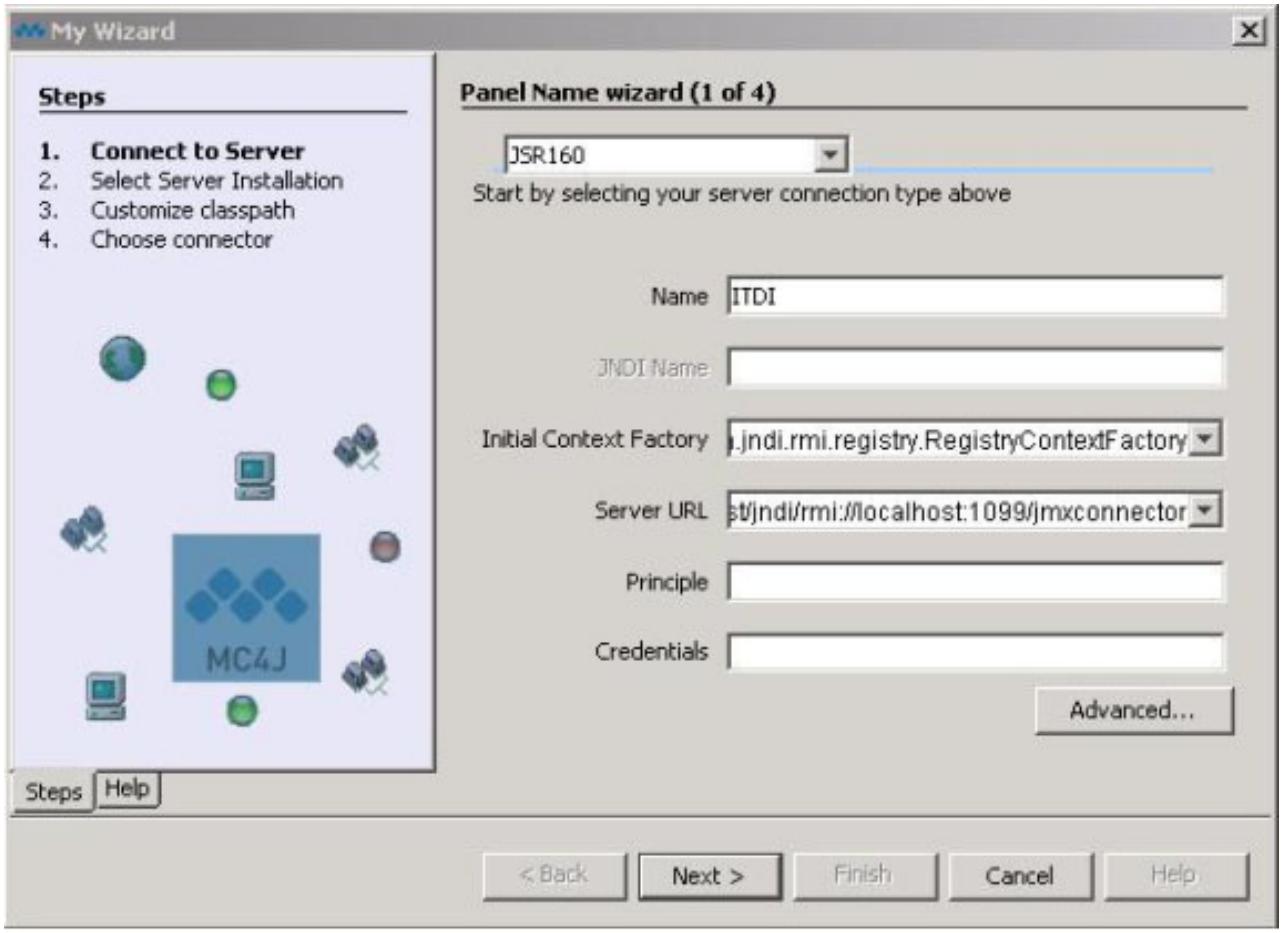


Figure 6.

3. Enter **TDI** in *Name* field.
4. In the **Server URL** text box paste the JMX connection URL dumped by the TDI server on startup

   Note: If TDI and MC4J are on different machines replace localhost with the TDI machine IP address.
5. Select **Next**.

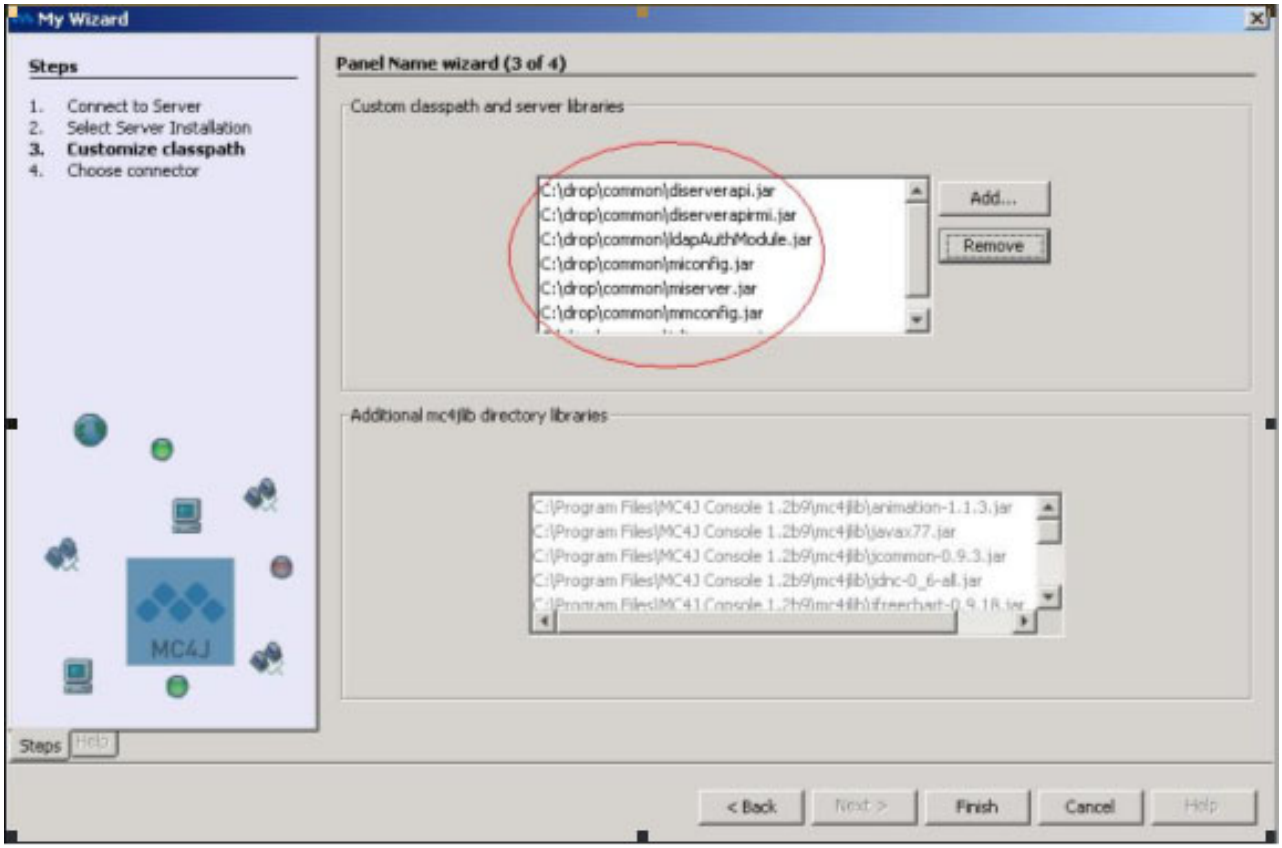


Figure 7.

6. In the **Custom classpath and server libraries** list, add all JAR files from the <*TDI_install_dir*>\jars\common folder.
7. Add these three jars as well:
   - <*TDI_home*>\jars\3rdparty\others\log4j-1.2.15.jar
   - <*TDI_home*>\jars\3rdparty\IBM\icu4j_4_2.jar
   - <*TDI_home*>\jars\3rdparty\IBM\ITLMToolkit.jar
   - Select **Finish**.

Now MC4J is connected to the TDI server.

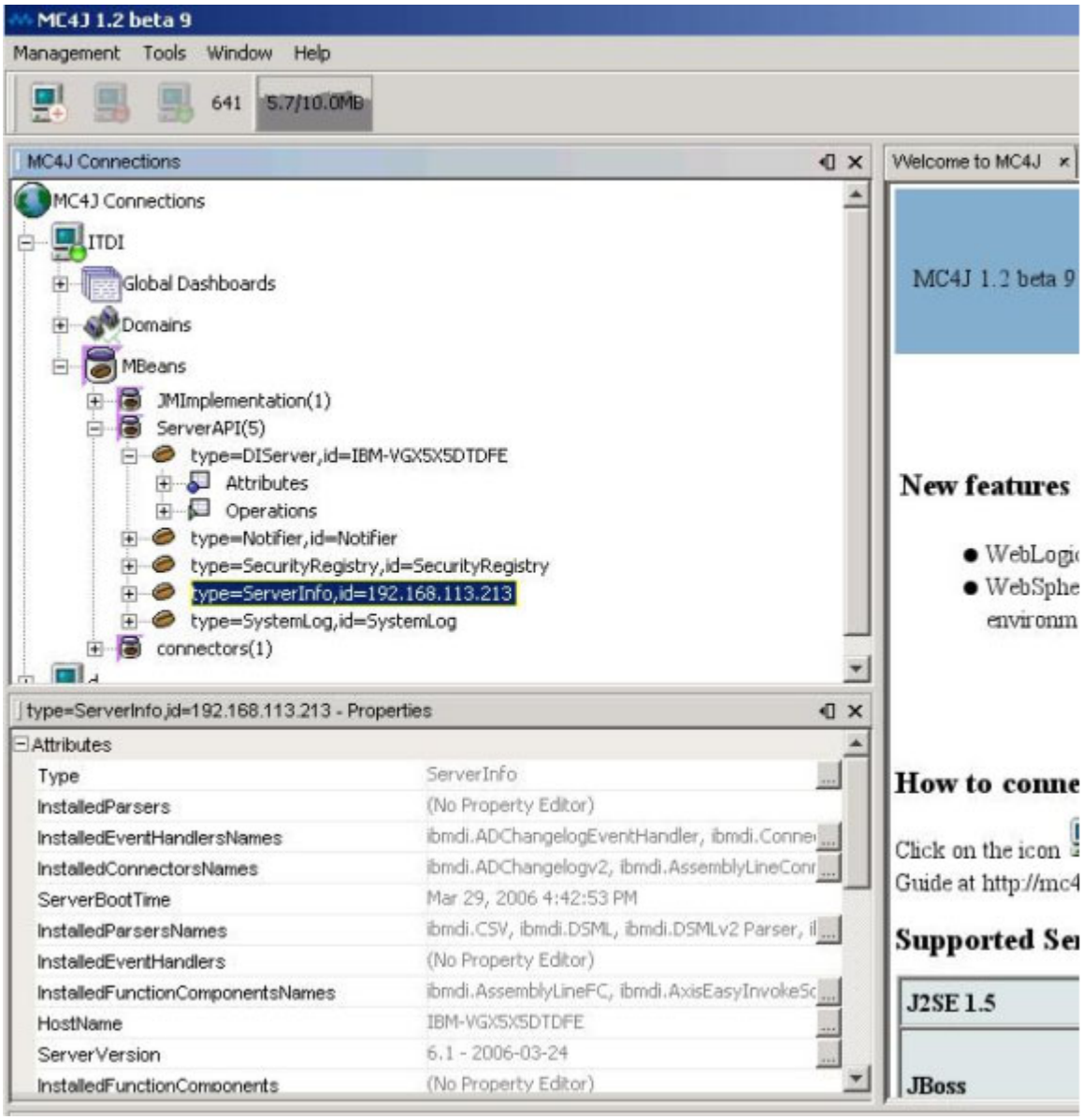


Figure 8.

Backward compatibility

Scenarios overview

While upgrading your TDI 6.0 installation to Tivoli Directory Integrator 7.1, you may find yourself in one of the following scenarios:

Table 77. Compatibility matrix

TDI Server version► **Client version▼**	**6.0**	**6.1**
6.0	OK	In most cases porting the client to 6.1 is required – see the "“Guidelines for porting a TDI 6.0 Server API client to use a Tivoli Directory Integrator 7.1 server”" section
6.1	OK with some caveats – see the "“Guidelines for implementing a Server API client capable of working with both TDI 6.0 and Tivoli Directory Integrator 7.1 servers”" section	OK

Guidelines for porting a TDI 6.0 Server API client to use a Tivoli Directory Integrator 7.1 server

When is porting required: Probably the most significant change in the Tivoli Directory Integrator 7.1 Server API is the way configurations are edited. For more information on editing configurations in Tivoli Directory Integrator 7.1 see the "Using the Server API -> Editing Configurations" section.

Server API changes in Tivoli Directory Integrator 7.1 relevant to porting a TDI 6.0 Server API client:

- There is a slight behavior change to a configuration editing related method described in "Table 80 on page 571 - Changed Methods".
- If a TDI 6.0 client uses some of the method calls listed in "Table 81 on page 571 - Deprecated methods" it needs to be reworked to use the new Tivoli Directory Integrator 7.1 methods instead.

Another reason to rework a TDI 6.0 client is to benefit from the functionality introduced in Tivoli Directory Integrator 7.1:

- There are several interfaces introduced in Tivoli Directory Integrator 7.1 described in "Table 78 on page 569 – New Server API interfaces".
- Some TDI 6.0 interfaces have been added new methods. A list of the new methods can be found in "Table 79 on page 569 - New methods".

Another important consideration while porting a TDI 6.0 client is the usage of serializable classes. More details can be found in the "“Using serializable classes” on page 568" section. A major part of the serializable classes used by the Server API are the TDI config interface classes. New serializable classes are listed in "Table 82 on page 571 - New Serializable classes/interfaces". A complete reference of the config interfaces can be found in the JavaDocs provided with TDI.

When is porting optional: If the TDI 6.0 client does not use config editing and there is no requirement to use the new Tivoli Directory Integrator 7.1 Server API features the TDI 6.0 client does not need to be modified.

Guidelines for implementing a Server API client capable of working with both TDI 6.0 and Tivoli Directory Integrator 7.1 servers

Since the enhancements in the Server API are done in a backward compatible manner it is possible a Server API client application to use all TDI 6.0 features against a TDI 6.0 Server and also use all new Tivoli Directory Integrator 7.1 features against a Tivoli Directory Integrator 7.1 Server. This can be accomplished by having the Server API client check the TDI server version and then execute the appropriate version specific code accordingly. An example is available in the "“Checking the TDI server version” on page 568" section.

There are two primary ways of sharing data between the Server API client and the TDI server:

- Using RMI remote objects
- Using serializable classes

Using RMI remote objects: In this case the Server API client will use remote object stubs generated from the Tivoli Directory Integrator 7.1 version of the remote classes. These stubs contain all methods existing in the TDI 6.0 version of the remote classes as well as the methods introduced in Tivoli Directory Integrator 7.1 (as they are described in "Table 79 on page 569 - New Methods"):

- The methods introduced in Tivoli Directory Integrator 7.1 cannot be used against a TDI 6.0 server. It is the responsibility of the client Server API application not to use these new methods against a TDI 6.0 server by checking the server version beforehand.
- The methods described in "Table 81 on page 571 - Deprecated methods" can only be used with a TDI 6.0 server. If a deprecated method is invoked on a Tivoli Directory Integrator 7.1 server an exception will be thrown.

Using serializable classes: The Server API serializable classes as well as the TDI serializable classes have evolved from TDI 6.0 to Tivoli Directory Integrator 7.1. Thus these classes are different in TDI 6.0 and in Tivoli Directory Integrator 7.1. Nevertheless these classes have evolved in a backward compatible way from a serialization perspective. This means that the TDI 6.0 serializable classes can interoperate with the Tivoli Directory Integrator 7.1 serializable classes through the Java RMI engine.

The Java RMI engine determines whether serializable classes are compatible by checking the class serial version UID – if the class serial version UID of two classes are identical then the RMI engine considers these two classes compatible. The serial version UIDs of the serializable classes in TDI can be found in "Table 83 on page 572 – serialVersionUID for serializable classes". Since the Tivoli Directory Integrator 7.1 serializable classes are compatible with the TDI 6.0 serializable classes the serial version UIDs of these classes have not changed.

"Table 82 on page 571 - New Serializable classes/interfaces" lists classes and interfaces introduced in Tivoli Directory Integrator 7.1. Since these classes and interfaces are not available in TDI 6.0 they cannot be used against a TDI 6.0 server. The TDI JavaDocs should be referred to for more detailed information on method signature changes of serializable classes in both releases. Methods which are not available in TDI 6.0 cannot be used against a TDI 6.0 server.

If the Server API client uses third party or custom user serializable classes then the best approach would be to ensure that these classes are identical on the server and on the client. If for any reason the serializable classes are different (but compatible) versions of the same class then the client still can work if both versions are set the same serialVersionUID. More information on maintaining and evolving serializable classes can be found at:

http://java.sun.com/j2se/1.5.0/docs/api/java/io/Serializable.html
http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serialTOC.html
http://www-03.ibm.com/developerworks/blogs/page/woolf?entry=serialization_and_serial_version_uid

Config Editing: Config editing in Tivoli Directory Integrator 7.1 is very different from config editing in TDI 6.0. That is why special care must be taken when coding editing TDI configs for both TDI 6.0 and Tivoli Directory Integrator 7.1 servers. This code is TDI version-specific. That is why the code needs to be branched by checking the TDI server version as described in the "“Checking the TDI server version”" section.

Authentication mechanisms: The username/password based authentication mechanism and the LDAP authentication mechanism are not supported on TDI 6.0. That is why the createSession (String aUserName, String aPassword) method of the com.ibm.di.api.remote.SessionFactory interface will fail if invoked against a TDI 6.0 server.

Checking the TDI server version: Usually most of the Server API client code will be common for TDI 6.0 and Tivoli Directory Integrator 7.1 servers. Sometimes, however, TDI 6.0- or Tivoli Directory Integrator

7.1-specific code could be needed. These version-specific portions of code require checking the server version. Below is a code sample which demonstrates how the TDI server version can be retrieved and used.

```
import com.ibm.di.api.remote.Session;
import com.ibm.di.api.remote.ServerInfo;

...

ServerInfo serverInfo = session.getServerInfo();
if (serverInfo == null) {
    throw new Exception("Server version information is not available!");
}

String serverVersion = serverInfo.getServerVersion();
if (serverVersion.startsWith("6.1")) {
    // TDI 6.1 specific code
}
else if (serverVersion.startsWith("6.0")) {
    // TDI 6.0 specific code
}
else {
    throw new Exception("Unsupported TDI server version: " + serverVersion);
}
```

Server API changes in Tivoli Directory Integrator 7.1

Table 78. New Server API interfaces

Name	Description
SystemQueue	Server API access to SystemQueue
TDIProperties	Wrapper for External Property Stores
TombstoneManager	Access to Tombstones read and delete

Table 79. New Methods

Name	Description
AssemblyLine	
String getGlobalUniqueID ()	Returns AssemblyLine GUID. The GUID is a string value that is unique for each component ever created by a particular TDI Server.
ConfigInstance	
String getGlobalUniqueID ()	Returns the Config Instance GUID. The GUID is a string value that is unique for each component ever created by a particular TDI Server.
String[] getConnectorPoolNames ()	Returns the names of all Connector Pools in the Config Instance.
int getConnectorPoolSize (String aConnectorPoolName)	Returns the size of the specified Connector Pool.
int getConnectorPoolFreeNum (String aConnectorPoolName)	Returns the number of free Connectors in the specified Connector Pool.
PoolDefConfig getConnectorPoolConfig (String aConnectorPoolName)	Returns the Connector Pool configuration object.
int purgeConnectorPool (String aConnectorPoolName)	Unused Connectors will be released so that the Pool is shrunk to its minimum size.
TDIProperties getTDIProperties()	Returns the TDIProperties object associated with the current configuration.
Session	

Table 79. New Methods (continued)

Name	Description
void shutDownServer (int aExitCode)	Shuts down the TDI Server with the specified exit code.
TombstoneManager getTombstoneManager ()	Returns the TombstoneManager object. Tombstones can be queried and cleared through this object.
boolean isSSLon ()	Checks if current session is over SSL.
boolean releaseConfigurationLock(String aRelativePath)	Administratively releases the lock of the specified configuration. This call can be only executed by users with the admin role.
boolean undoCheckOut(String aRelativePath)	Releases the lock on the specified configuration, thus aborting all changes being done. This call can only be executed from a user that has previously checked out the configuration and only if the configuration lock has not timed out.
ArrayList listConfigurations(String aRelativePath)	Returns a list of the file names of all configurations in the specified folder. The configurations file paths returned are relative to the Server configuration codebase folder.
ArrayList listFolders(String aRelativePath)	Returns a list of the child folders of the specified folder.
ArrayList listAllConfigurations()	Returns a list of the file names of all configurations in the directory subtree of the Server configuration codebase folder. The configurations file paths returned are relative to the TDI Server configuration codebase folder.
MetamergeConfig checkOutConfiguration (String aRelativePath)	Checks out the specified configuration. Returns the MetamergeConfig object representing the configuration and locks that configuration on the Server.
MetamergeConfig checkOutConfiguration (String aRelativePath, String aPassword)	Checks out the specified password protected configuration. Returns the MetamergeConfig object representing the configuration and locks that configuration on the Server.
ConfigInstance checkOutConfigurationAndLoad (String aRelativePath)	Checks out the specified configuration and starts a temporary Config Instance on the Server.
ConfigInstance checkOutConfigurationAndLoad (String aRelativePath, String aPassword)	Checks out the specified configuration and starts a temporary Config Instance on the Server.
void checkInConfiguration (MetamergeConfig aConfiguration, String aRelativePath)	Saves the specified configuration and releases the lock. If a temporary ConfigInstance has been started on check out, it will be stopped as well.
void checkInAndLeaveCheckedOut (MetamergeConfig aConfiguration, String aRelativePath)	Checks in the specified configuration and leaves it checked out. The timeout for the lock on the configuration is reset.
void checkInConfiguration (MetamergeConfig aConfiguration, String aRelativePath, boolean aEncrypt)	Encrypts and saves the specified configuration and releases the lock. If a temporary Config Instance has been started on check out, it will be stopped as well.
MetamergeConfig createNewConfiguration (String aRelativePath, boolean aOverwrite)	Creates a new empty configuration and immediately checks it out. If a configuration with the specified path already exists and the aOverwrite parameter is set to false the operation will fail and an Exception will be thrown.

Table 79. New Methods (continued)

Name	Description
ConfigInstance createNewConfigurationAndLoad (String aRelativePath, boolean aOverwrite)	Creates a new empty configuration, immediately checks it out and loads a temporary Config Instance on the Server. If a configuration with the specified path already exists and the aOverwrite parameter is set to false the operation will fail and an Exception will be thrown.
boolean isConfigurationCheckedOut (String aRelativePath)	Checks if the specified configuration is checked out on the Server.
void sendCustomNotification (String aType, String aId, Object aData)	Sends a custom, user defined notification to all registered listeners.
SystemQueue getSystemQueue()	Gets the remote Server API SystemQueue representation object
String getConfigFolderPath()	Gets the value of the api.config.folder property in the remote server as a complete path. If not set, then returns an empty string.
Object invokeCustom (String aCustomClassName, String aMethodName, Object[] aParams)	Invokes the specified method from the specified class.
Object invokeCustom (String aCustomClassName, String aMethodName, Object[] aParamsValue,String[] aParamsClass)	Invokes the specified method from the specified class.
SessionFactory	
Session createSession (String aUserName, String aPassword)	Creates a session object with the specified username and password.

Table 80. Changed Methods

Name	Description
ConfigInstance	
void setConfiguration (MetamergeConfig aConfiguration)	In Tivoli Directory Integrator 7.1 this method can be invoked only if a particular client has already checked out same config with temporary config instance.

Table 81. Deprecated methods (these are methods which are not to be used against a Tivoli Directory Integrator 7.1 server; it is perfectly OK to use these methods against a TDI 6.0 server)

Name	Description
ConfigInstance	
void saveConfiguration ()	Use CheckIn methods instead of save
void saveConfiguration (boolean aEncrypt)	Use CheckIn methods instead of save
void setExternalProperties (ExternalPropertiesConfig aExPropConfig)	Use TDIProperties
void setExternalProperties (String aKey, ExternalPropertiesConfig aExPropConfig)	Use TDIProperties
void saveExternalProperties ()	Use TDIProperties

Table 82. New Serializable classes/interfaces

Name	Description
com.ibm.di.api.Tombstone	5178569311755396746L
com.ibm.di.api.CIEvent	5178569311755396746L

Table 82. New Serializable classes/interfaces (continued)

Name	Description
com.ibm.di.config.interfaces.NamespaceEvent	-1857414661726671152L
com.ibm.di.config.interfaces.OperationConfig	2715909691453046036L
com.ibm.di.config.interfaces.PoolDefConfig	-1252371938517765606L
com.ibm.di.config.interfaces.PoolInstanceConfig	5594919717769030291L
com.ibm.di.config.interfaces.PropertyManager	4280805548502266432L
com.ibm.di.config.interfaces.PropertyStoreConfig	-2620929677558833640L
com.ibm.di.config.interfaces.ReconnectConfig	-7935628947261477628L
com.ibm.di.config.interfaces.TDIProperties	-33614718378886772777L
com.ibm.di.config.interfaces.TDIPropertyStore	198251115520372634L
com.ibm.di.config.interfaces.TombstonesConfig	-3260102686391332434L

Table 83. serialVersionUID for serializable classes

Name	Status	serialVersionUID
com.ibm.di.api.ALEvent	backward compatible	5631772256973692972L
com.ibm.di.config.base.ALMappingConfigImpl	backward compatible	2712493657450710788L
com.ibm.di.server.ALState	backward compatible	669938312260868491L
com.ibm.di.config.base.AssemblyLineConfigImpl	backward compatible	2715909691453046036L
com.ibm.di.entry.Attribute	backward compatible	6675881744901860329L
com.ibm.di.config.base.AttributeMapConfigImpl	backward compatible	-26190155381786656841L
com.ibm.di.entry.AttributeValue	backward compatible	100100L
com.ibm.di.config.base.BaseConfigurationImpl	known issue – see the "Known issues" on page 573 section	-73169799792531250051L
com.ibm.di.config.base.BranchConditionImpl	backward compatible	-40917732335838179121L
com.ibm.di.config.base.BranchingConfigImpl	backward compatible	-10135888843811339441L
com.ibm.di.config.base.CallConfigImpl	backward compatible	-46974584978353290961L
com.ibm.di.config.base.CallParamConfigImpl	backward compatible	57880211547147417671L
com.ibm.di.config.base.CheckpointConfigImpl	backward compatible	-83423698815234684831L
com.ibm.di.config.base.ConfigCache	backward compatible	-33112557315041744161L
com.ibm.di.config.base.ConfigStatistics	backward compatible	-1271645457384911249L
com.ibm.di.config.base.ConnectorConfigImpl	backward compatible	4093376456212230000L
com.ibm.di.config.base.ConnectorSchemaConfigImpl	backward compatible	930161291800752910L
com.ibm.di.config.base.ConnectorSchemaItemConfigImpl	backward compatible	-1665598194757295769L
com.ibm.di.config.base.ContainerConfigImpl	backward compatible	-4134004409592694052L
com.ibm.di.config.base.DeltaConfigImpl	backward compatible	-72501284845880240171L
com.ibm.di.api.DIEvent	backward compatible	-86645334774524912191L
com.ibm.di.entry.Entry	backward compatible	-5961424529378625729L
com.ibm.di.config.interfaces.ExternalPropertiesDelegator	known issue – see the "Known issues" on page 573 section	7725187425731381660L
com.ibm.di.config.base.ExternalPropertiesImpl	backward compatible	-5837658758525300221L

Table 83. serialVersionUID for serializable classes (continued)

Name	Status	serialVersionUID
com.ibm.di.config.base.FormConfigImpl	backward compatible	-8761349695805705052L
com.ibm.di.config.base.FormItemConfigImpl	backward compatible	-7825109041707716857L
com.ibm.di.config.base.FunctionConfigImpl	backward compatible	5778585850194005910L
com.ibm.di.config.interfaces.GlobalRef	backward compatible	366178307603105225L
com.ibm.di.config.base.HookConfigImpl	backward compatible	-1300997546910640256L
com.ibm.di.config.base.HooksConfigImpl	backward compatible	-9160883008989377612L
com.ibm.di.config.interfaces.InheritanceLoopException	backward compatible	-5977834080357995975L
com.ibm.di.config.base.InheritConfigImpl	backward compatible	90155321639831994871
com.ibm.di.config.base.InstanceConfigImpl	backward compatible	-7052997089129596762L
com.ibm.di.config.base.LibraryConfigImpl	backward compatible	-6737181973806281819L
com.ibm.di.config.base.LinkCriteriaConfigImpl	backward compatible	-92068565361720118211
com.ibm.di.config.base.LinkCriteriaItemImpl	backward compatible	-952539248920610452L
com.ibm.di.config.base.LogConfigImpl	backward compatible	3371411072185625170L
com.ibm.di.config.base.LogConfigItemImpl	backward compatible	62997504647888089711
com.ibm.di.config.base.LoopConfigImpl	backward compatible	-81745410745104814181
com.ibm.di.config.base.MetamergeConfigImpl	backward compatible	-336369533068596790 4L
com.ibm.di.config.xml.MetamergeConfigXML	backward compatible	-4403169711579029765L
com.ibm.di.config.base.MetamergeFolderImpl	backward compatible	6107586753523140220L
com.ibm.di.config.base.NamespaceConfigImpl	backward compatible	986964857890827079L
com.ibm.di.config.base ParserConfigImpl	backward compatible	5497221494799800099L
com.ibm.di.config.base.PropertyConfigImpl	backward compatible	-26209296775588336401
com.ibm.di.config.base.RawConnectorConfigImpl	backward compatible	8439049716964119460L
com.ibm.di.config.base.SandboxConfigImpl	backward compatible	-399320124155373314L
com.ibm.di.config.base.SchemaConfigImpl	backward compatible	1778816095104785134L
com.ibm.di.config.base.SchemaItemConfigImpl	backward compatible	51688019478113765661
com.ibm.di.config.base.ScriptConfigImpl	backward compatible	-77476862425517938901
com.ibm.di.api.remote.impl.rmi.SSLRMIClientSocketFactory	backward compatible	50830175460314203841
com.ibm.di.server.TaskCallBlock	backward compatible	115072761837771375L
com.ibm.di.server.TaskStatistics	backward compatible	2098518046376889585L
com.ibm.di.api.remote.impl.rmi.RMISocketFactory	backward compatible	-3200652858929712303L

Known issues

com.ibm.di.config.interfaces.ExternalPropertiesDelegator

The com.ibm.di.config.interfaces.ExternalPropertiesDelegator class is the implementation class of the com.ibm.di.config.interfaces.ExternalPropertiesConfig interface. The com.ibm.di.config.interfaces.ExternalPropertiesDelegator class also extends the com.ibm.di.config.base.BaseConfigurationImpl class.

Server API client code deals with interfaces and not classes, that is why the ExternalPropertiesDelegator class is not directly referenced in the Server API client source code. The limitation is that while a TDI 6.0 client can retrieve an ExternalPropertiesConfig object from a TDI 6.0 server, this client cannot modify the

external properties on the server by calling the setExternalProperties(String aKey, ExternalPropertiesConfig aExPropConfig) or the setExternalProperties(ExternalPropertiesConfig aExPropConfig) on a config instance object (com.ibm.di.api.remote.ConfigInstance). If one of these methods is invoked from a Tivoli Directory Integrator 7.1 client against a TDI 6.0 server it will fail.

com.ibm.di.config.base.BaseConfigurationImpl

The same issue as the above one discussed for com.ibm.di.config.interfaces.ExternalPropertiesDelegator applies to com.ibm.di.config.base.BaseConfigurationImpl as well. (The former is an extension of the latter.)

Appendix D. Creating new components using Adapters

Introduction

The Adapter concept is a mechanism in Tivoli Directory Integrator to enable developers to create new custom Connectors by using the AssemblyLine (AL) methodology. The alternative would be to develop these in either Java or JavaScript. Adapters are easy to distribute to other Tivoli Directory Integrator developers, and just as simple to use as traditional, pre-built standard Connectors.

Adapters enable developers to leverage the entire Tivoli Directory Integrator arsenal when creating custom connector with potentially complex business logic and custom operations to be offered to the Tivoli Directory Integrator development community.

There are a number of new features that in combination make Adapters possible that will be described in the sections below. All of these features have value outside the Adapter concept as well, so we advise you to read about each feature in the rest of the formal documentation.

The following is the high level flow of activities to implement and use a Tivoli Directory Integrator Adapter:

1. Anne develops the Adapter AssemblyLine (for example to access a custom developed ERP system) that implements the connector modes to be supported (such as iterator and delete), as well as custom modes as required.
2. Anne publishes the AL into a package that can be distributed to Pete, another Tivoli Directory Integrator developer, as a stand-alone file.
3. Pete copies the package into his Tivoli Directory Integrator development environment. The resource/library model is ideal for this purpose, along with other components that Pete wants to re-use across the Tivoli Directory Integrator solutions that he routinely develops.
4. Pete uses Anne's Adapter in his AL just like any other Tivoli Directory Integrator connector by using an “AssemblyLine Connector” on page 14 to call the Adapter.
5. Anne can improve her Adapter and publish new versions by going through the steps above.

The picture below illustrates Pete's AL on the left using Anne's Adapter both in standard Connector *lookup* mode, as well as the custom *Disable_acct* mode. In the illustration, even though the Adapter is used two places in Pete's AL, there is only one instance started of the Adapter to reduce the impact on the back-end target system. Just like normal Connectors, Adapters can be shared within an AL, pooled – and even shared – across AL's.

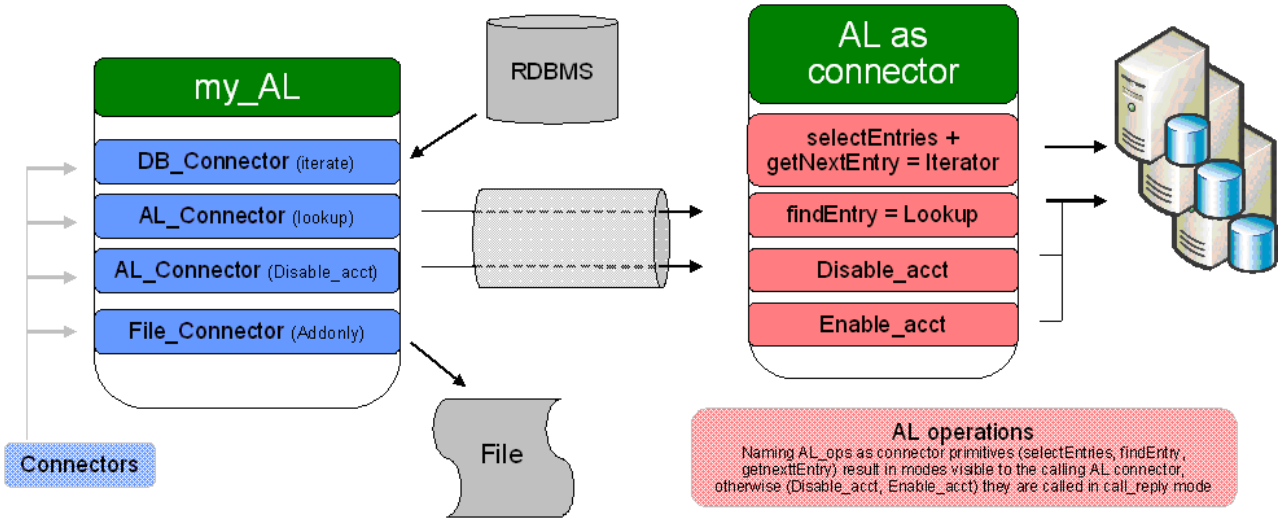


Figure 9. Overview of Adapter usage

Features that enable implementation of a Tivoli Directory Integrator Adapter

We shall look a little deeper at the features that are utilized to create and use Adapters. Most of these features are not developed specifically for the Adapter concept, so they have many use cases for non-Adapter use as well.

AL Operations

Any number of *operations* can be defined in an AL. They are similar to the pre-7.0 call-return schema of the AL (this is now the so-called "Default" operation), but any number of operations can now be created. When AL's are called/executed through the API, from script, from the AL FC, or from the AL Connector, an operation may now be specified along with the required attributes for that operation. At run-time, the AL will know what operation has been invoked; this can be queried, and the flow inside the AL can be adjusted accordingly.

Note: When you define operations for an AssemblyLine, then unless you also define a "Default" operation, you are now obliged to invoke the AssemblyLine specifying an explicit operation, otherwise the AssemblyLine will throw an exception.

These operations provide the entry points for the AL Connector (described below in the section “Using an Adapter in your AssemblyLine” on page 578) to view and treat the Adapter as a Connector. The entry points are the same as when developing a Connector in Java or JavaScript, and are described in Appendix E, “Implementing your own Components in Java,” on page 583 as well as in a table in the section “Mapping Adapter operations to Connector modes” on page 578. For example, if the Adapter developer only wants to implement the "lookup" mode, then it's only necessary to implement the "findEntry" operation. More details are found in section “The use of operations in a Tivoli Directory Integrator Adapter” on page 578 below.

Switch/case component

The *Switch* component is an AL component that is similar to switch constructs in traditional development languages. Basically, it's a variant of the If-ElseIf-ElseIf component. Within the Switch component, a number of Case's are defined that contain the AL components to be executed when the Switch statement matches the value of the case. One benefit of the Switch component is that it can automatically populate

the case statements based on the AL operations that have been defined. That way one can easily ensure that code is implemented for all of the operations that have been defined.

Flexible connector initialization

In older versions of Tivoli Directory Integrator, all Connectors in an AL got initialized during the AL initialization phase. Dynamic configuration of a connector usually required termination of a connection, modification of the connection parameters, and then a re-establishment of the connection – all through script. The alternative was to establish and use the connector solely through script.

In the current version of Tivoli Directory Integrator, Connectors can optionally initialize:

On demand
: Affectionately knows as "lazy". The connection is established not at AL initialization, but as control the first time actually passes to this connector during the AL execution phase (Flow). This means that a complex AL will only initialize the connectors that are actually used.

Every time
: The Connector initializes every time as control is passed to it. This is useful when the connection parameters (such as a file name or LDAP credentials) are part of the information passed into each call to the Adapter. A separate benefit is that this capability is also helpful when using pooled connectors, as "every time" will result in acquiring a connector instance from the pool at run-time, and then released after use. In essence, this implements a "shared/re-use" connector pool across AL's.

When config has changed
: The Connection is re-initialized if the config parameters (or evaluations of parameter substitution) have changed since the previous initialization of the Connector. With the parameter substitution feature, connectors can be dynamically configured much easier than before. This facilitates changing the connection parameters of a connector – and forcing re-connection – from both inside the AL as well as outside. For example, another AL, or a command-line modification of properties can result in an AL automatically re-connecting to its targets with little effort.

Using an Iterator in Flow

Iterators have previously only been used in the Feed section to drive the entire AL cycle, or within the Flow to power a Loop component. Now an iterator can be placed in the Flow itself to facilitate implementing Iterator mode in an Adapter. To understand this, a short review of how the iterator works is in order. First selectEntries() of a Connector (or for Adapters, the selectEntries operation as described below in "The use of operations in a TDI Adapter"), is called to create the result set, then getNextEntry() is called to read from the result set until it's empty. By limiting iterators to the Feed section, it would be very impractical to implement a getNextEntry operation that returned the next record from an iterator Connector in your Adapter. With an iterator in the Flow, your getNextEntry operation could utilize an iterator connector and make life a lot easier.

Packing an Adapter for consumption

The goal, in the scenario depicted in the introduction, would be to help Anne package her Adapter component and publish it for others to consume.

When the Adapter has been developed, the Publish command in the Tivoli Directory Integrator development environment creates a Package of Anne's AL. Publishing an AL means resolving all inheritance and dependencies between the Adapter AL and the rest of Anne's development environment. The Package consists of a standard stand-alone config XML file that only contains the Adapter code that can be sent to other TDI developers for inclusion in their resource library (more below).

The package can be saved anywhere, but the default location is the packages directory in Anne's Tivoli Directory Integrator solution directory. When the Adapter is published, it shows up in the Adapter

section of the resources library, in the connector list as an available connector, and can also be queried from the AL Connector (see section “Using an Adapter in your AssemblyLine”).

At this stage you might be a bit confused about the difference between a Package and an Adapter. Basically, Adapters are Packages that are intended to be used as Connectors. Other Packages might just contain ALs that can be called with the AL FC or other mechanism to run an AL.

Using an Adapter in your AssemblyLine

The goal is to provide the interfacing mechanism so that Pete can utilize Anne's Adapter in his own AL's as any other Connector.

There are a number of mechanisms available when calling an AL from another AL. However, when an AL has been developed as an Adapter, then the primary mechanism to use is the “AssemblyLine Connector” on page 14 (AL Connector). This Connector can deal with Adapter-style ALs. In older versions (6.0 and earlier) of Tivoli Directory Integrator, it could only be used to iterate on the output of another AL. Currently, when the AL Connector is used in an AssemblyLine, it is configured by specifying what Adapter it should call. The target Adapter is then inspected for operations, and that determines what Connector modes are made available to the developer.

As a convenience feature, Tivoli Directory Integrator automatically wraps all Adapters so that they look like connectors in the connector list. Pete can therefore choose to insert an Adapter directly from his connector list, or can insert an AL Connector and then specify the desired Adapter to call.

The configuration of the Adapter is done in the usual Connector config panel. All parameters displayed here are defined in the schema of the reserved operation "$initialization" of the Adapter. This provides the Tivoli Directory Integrator developer with a mechanism to send configuration parameters to the Adapter for dynamic configuration of its Connectors.

The “Flexible connector initialization” on page 577 is useful here, in that it can be used both in the AL that calls the Adapter, as well as in the Adapter itself. Using "on demand" or "every time" initialization of the Adapter, the calling AL can use information retrieved during its execution phase to configure the Adapter, rather than having to pre-configure this through more static mechanisms.

The use of operations in a Tivoli Directory Integrator Adapter

To implement the Tivoli Directory Integrator Connector modes, AL operations have to be created in the Adapter that correspond to the Connector primitives that any Connector has to implement, just as if it was implemented in Java or JavaScript.

The AL Connector will automatically determine what modes are available by inspecting what operations that have been defined in the Adapter. It is only necessary to implement the operations that correspond to the modes that you want the Adapter to expose. Operations that do not correspond to any in the table below are exposed as additional Adapter modes, and executed in call/reply mode by the AL Connector.

Mapping Adapter operations to Connector modes

The following are the methods that a developer has to consider when implementing a Tivoli Directory Integrator connector in Java or JavaScript. Please refer to Appendix E, “Implementing your own Components in Java,” on page 583 to fully understand the relationships between these methods and the Connector modes that they implement. For example, to implement Lookup mode in an Adapter, only the findEntry operation needs be defined.

Table 84. Operations versus modes

	Iterator	Lookup	AddOnly	Update	Delete	Delta[2]	CallReply	Server[3]
initialize	(X)	(X)	(X)	(X)	(X)	(X)	(X)	(X)

Table 84. Operations versus modes (continued)

	Iterator	Lookup	AddOnly	Update	Delete	Delta[2]	CallReply	Server[3]
querySchema	(X)	(X)	(X)	(X)	(X)	(X)	(X)	(X)
selectEntries	(X)							
getNextEntry	X							X
findEntry		X		X	X	(X)		
modEntry				X		X		
putEntry			X	X		X		(X)
deleteEntry					X	X		
queryReply							X	
getNextClient								X
terminate	(X)	(X)	(X)	(X)	(X)	(X)	(X)	(X)

Notes:

1. (X) means optional. Will be called if they exist.
2. Delta mode is handled in a special manner. See section below.
3. Server mode is not currently supported in Adapters.
4. The operation names are case sensitive.

Implementing code in the Adapter for each operation

The Adapter AL is called by the "AssemblyLine Connector" on page 14 according to the rules for the modes that are exposed – through the operations illustrated in the table above. The Adapter must check for what operation has been invoked and execute the corresponding code in the AL. The Switch Component is well suited for this purpose, but If-Else Components can be used as well by creating conditions using the op-entry.$operation attribute which will be set each time the AL is called with an operation. This attribute may of course be used in a script as well.

Op-entry is an Entry object available for use in Adapter ALs. Like the *work* object it is created by the AssemblyLine, but it doesn't get cleared every time the AL cycles. It is used to store attributes that the AL needs throughout its lifecycle. The next section will show further use of it.

Adapter configuration through the $initialization operation

All attributes that are defined in the schema of the $initialization operation of the Adapter are displayed in the configuration panel of the AL Connector that calls the Adapter. These attributes are passed to the Adapter at initialization time so that the Adapter can perform the necessary preparation and connection to the target systems.

The attributes defined in the $initialization schema are available to the Adapter throughout its lifecycle as attributes in the op-entry attribute.

Connectors in the Adapter can be configured with these attributes by using expressions in the connector parameter fields. For example, if the Adapter has defined an *ou* attribute in its $initialization schema, then the user of the Adapter will see "ou" as one of the configuration parameters in the AL Connector. The Adapter could then define a search base in an LDAP connector as:

```
cn=...,ou={op-entry.ou}
```

These attributes will be available at the initialization time of the Adapter, which by default is the same time as the calling AL is initialized, unless one of the mechanisms described above in section "Flexible connector initialization" on page 577 is utilized.

Understanding the link criteria

The link criteria defined in the AL Connector is passed into the Adapter through the *search* object (*SearchCriteria*) in the op-entry object.

Extracting the individual criteria objects can be done with the following script code:

```
search = Task.getOpEntry().getObject("search");
criteria = search.getCriteria(0); /* index ranges from 0 to search.size() */
name  = criteria.name;      /* target attribute */
match = criteria.match;      /* expression (less, greater, equal.. */
value = criteria.value;      /* value to test the target attribute against through the expression */
negate = criteria.negate;      /* Boolean flag */
```

Each criteria object contains the attributes: name, match, value, and negate (boolean).

The *search* object provides convenience methods to create LDAP, Domino and SQL search strings based on its link criterias. Please refer to the Javadocs for further information.

Attribute mapping

When the Adapter operations are called through the AL Connector, the work Entry is populated with the attributes in the output map of the AL Connector. On return, the AL Connector expects returned attributes either in *work*, or in the *conn* object as described in the next sections.

From the calling AL into the Adapter

The Adapter must use script methods such as:

```
email = work.getString("email");
```

to extract the value of the email attribute so that it can be used in further attribute mapping inside the Adapter. A practical suggestion is to insert an AL level Attribute Map (Attmap) component early in the Adapter to extract the desired attributes from *conn* and make them visible in *work* for easy reference in the rest of the Adapter.

Return data from the Adapter to the calling AL

The modes *iterator*, *lookup* and *callreply*, return data to the calling AssemblyLine and populate the output map of the AL Connector. The simplest way to return attributes to the calling AL is through the *work* object. Any attributes left in *work* at the end of the Adapter cycle will be passed back to the input map of the calling AssemblyLine Connector. It is therefore important to remove temporary work attributes at the end of the Adapter so that they aren't inadvertently returned as well, for example like this:

```
work.removeAttribute("attributeName");
...
```

The Adapter indicates end of data by returning an empty conn object in *work*. An empty *work* object is not sufficient since that is merely interpreted as an empty record by the AL Connector. To indicate end of data by the iterator, use

```
work.newAttribute("conn");
```

Lookup Mode:

The lookup mode may return multiple records. If it is necessary to return more than one record, the Adapter must create the Entry attribute *conn* in the work Entry that can contain zero, one, or more values of type Entry. Further on in this section there is some script code to illustrate how this can be achieved in an Adapter.

The following is a mix of JavaScript and pseudo-code to illustrate the part of implementing the *findEntry* operation (that implements *lookup* mode) where attributes are mapped into a structure that can be returned to the AL Connector in the calling AL.

The example illustrates two ways to return multiple records to the calling AL. The example on the right hand side is simpler because *work* is cleared for each iterator cycle, and all the values in work are therefore a result of the iterator's output map, and can therefore be added to acc (acc is shorthand for accumulator) in a single operation. An important note is that getClone() needs to be used to ensure that the value of the attributes are copied into *acc*.

	Clear work for each iterator cycle:

`acc = system.newEntry().newAttribute("conn");`
`Loop on iterator (that returns attributes a,b,c from target into work)`
`{`
`/* all of the below would be located in a script component inside the Loop component */`
`temp = system.newEntry();`
`temp.setAttribute(work.getAttribute("a"));`
`temp.setAttribute(work.getAttribute("b"));`
`temp.setAttribute(work.getAttribute("c"));`
`acc.addValue(temp) ;`
`}`
`work.setAttribute("conn", acc);`
	`work.removeAllAttributes();`
	`acc = system.newEntry().newAttribute("conn");`
	`Loop on iterator (that returns attributes a,b,c from target into work)`
	`{`
	`acc.addValue(work.getClone());`
	`work.removeAllAttributes();`
	`}`
	`work.setAttribute("conn", acc);`

Status indication

A good practice is to return the attribute *recordsProcessed* to indicate how many records were deleted, modified, or otherwise processed. This attribute can be passed back to the calling AL as in the *work* object. To indicate an error situation where the AL Connector should invoke one of the error hooks in the calling AL, the Adapter needs to throw an exception. Please refer to the section on error handling for more details on this.

Implementing Query Schema

A user of the Adapter will want to discover the schema of the Adapter. This is typically done when configuring the AL Connector where there are buttons to *connect* and to *query schema*. If the Adapter implements a static schema, then the simple solution is to create a *querySchema* operation in the Adapter, and define the schema there. The schema defined in the *querySchema* operation will be common for all standard Connector modes. Specific schemas can be defined for any non-standard modes. For example, if the Adapter implements an "AddUser" operation, then it can have its own schema defined.

Delta mode

Delta mode is handled somewhat differently from other modes because there are two different scenarios for handling delta data – meaning an Entry that has been tagged for change at the Entry, attribute and/or value level.

1. If the target system (or implemented in the Adapter) supports change based modification (for example, LDAP allows individual values to be updated in a specific attribute in a specific entry without supplying any of the other values of the attribute). These systems are defined as "Delta savvy" and indicate that the Adapter can deal with a tagged Entry.
2. For other systems, Delta mode can be simulated by performing either delete, add, or a sequence of find and then applying the proper changes to the record before writing the entire record back with modify. This is something that the AL Connector can do by using the basic Adapter primitives, but the Adapter needs to indicate that this is desired functionality.

To enable Delta behaviour in an Adapter, first the Delta operation needs to be defined. The next option is to create an attribute *deltaSavvy* in the Delta schema. Without the deltaSavvy attribute, the AL Connector will simulate the Delta mode as described above. With the deltaSavvy attribute in place, the AL Connector not call *findEntry* first, but rather call *modEntry* operation directly where it is the Adapters job to inspect the attributes for tags and apply the appropriate commands against the target system.

Error handling

Throw exceptions in your Adapter code to let the calling AssemblyLine drop the user into error hooks of the AL Connector, such as:

```
throw new java.lang.Exception ("error message");
```

Appendix E. Implementing your own Components in Java

This chapter is intended for developers that are tasked with creating new Connectors or Function Components for IBM Tivoli Directory Integrator (Tivoli Directory Integrator). They should have a firm understanding of Tivoli Directory Integrator operations as well as experience in developing with the Java language.

This material does not describe how to develop parsers, and assume that parsing logic is implemented in the component itself. A separate document will be provided to cover this theme.

Support materials for Component development

The `DirectoryConnector.java` file contains Java code which is a helpful examples when reading this tutorial. The file is located in the *root_directory/examples/connector_java* directory.

All java docs for the core Tivoli Directory Integrator classes cited in this chapter are located in the "`/docs/api`" folder of your Tivoli Directory Integrator installation. You can view this documentation by selecting the **Help -> Welcome** screen, **JavaDocs** link from within the Config Editor.

Developing a Connector

Implementing the Connector's Java source code

All Tivoli Directory Integrator Connectors implement the "*com.ibm.di.connector.ConnectorInterface*" Java interface. This interface provides a number of methods to implement addressing all the possible ways of using a Connector within Tivoli Directory Integrator. Usually the Connectors you write will not require all the options provided by Tivoli Directory Integrator and you will actually need to implement only a subset of the methods presented in the "*ConnectorInterface*" interface. It is the "*com.ibm.di.connector.Connector*" class that makes this possible.

"*com.ibm.di.connector.Connector*" is an abstract class implementing "*ConnectorInterface*" that contains core Connector functionality (for example processing of Connector's configuration) and also provides empty or default implementation to many of the methods from "*ConnectorInterface*". This allows you to start implementing your Connector by subclassing "*com.ibm.di.connector.Connector*" and focusing on (implementing) only those methods from "*ConnectorInterface*" that provide value in your case, and that are actually necessary for your Connector.

Listed below are the "*ConnectorInterface*" methods that build the backbone of a real Connector, and which you will usually need to implement:

Connector's constructor
: Required for all Connector modes.

  In the constructor you will usually set the name of your Connector (using the "*setName(...)*" method) and define what modes – Iterator, Lookup, AddOnly, Server, Delta etc. – that your Connector supports (using the "*setModes(...)*" methods). For an example of a Connector implementation, look at the "*DirectoryConnector.java*" Connector included in this package.

public void initialize (Object object)
: This method is called by the AssemblyLine before it starts cycling. In general anybody who creates and uses a Connector programmatically should call "*initialize(...)*" after constructing the Connector and before calling any other method.

Usually the "*initialize(...)*" method reads the Connector's parameters and makes the necessary preparations for the actual work (creates a connection, etc.) based on the parameter values specified.

public void selectEntries ()

Required for Iterator mode. This method is called only when the Connector is used in Iterator mode, after it has been initialized.

Place in "*selectEntries(...)*" any code you need to execute prior to actually starting to iterate over the Entries. When the Connector operates on a database, that code could be an **SQL SELECT** query that returns a result set; when the Connector operates on an LDAP directory, that code could be a search operation that returns a result set. The result of the "*selectEntries(...)*" (result set, etc.) is later used by the "*getNextEntry(...)*" method to return a single Entry on each call/AssemblyLine iteration. Of course you might not need any preparation to iterate over the Entries (as in the case with the FileSystem Connector) in which case there is no need to implement "*selectEntries(...)*". By subclassing "*com.ibm.di.connector.Connector*" you will inherit its default implementation that does nothing.

public Entry getNextEntry ()

Required for Iterator mode. This is the method called on each AssemblyLine's iteration when the Connector is in Iterator mode.

It is expected to return a single Entry that feeds the rest of the AssemblyLine.

There are no general guidelines for implementing this method – it all depends on the information this Connector is supposed to access. This method retrieves data from the connected data source and must create an Entry object and populate it with Attributes. For example, a database Connector would read the next record from a table/result set and build an Entry object whose Attributes correspond to the record's fields.

public Entry findEntry (SearchCriteria search)

Required for Lookup, Update and Delete modes. It is called once on each AssemblyLine iteration when the Connector performs a Lookup operation.

This method finds matching data in the connected system based on the "Link Criteria" specified in the Config Editor GUI. For example, a database Connector would execute a **SELECT** query with the appropriate **WHERE** clause based on Link Criteria and then build an Entry from the database record, in the same way as "*getNextEntry()*" does. Please consult the Java Docs for the structure of the SearchCriteria input parameter.

- When the specified link criteria succeeds in finding exactly one Entry, it should return that Entry.
- When the specified link criteria results in either zero or multiple Entries found (that is, anything but a single match), the method should return **NULL**. However, in the case of a multiple entry match, it must still provide the entries found so that they can be accessed from the "On Multiple Entries" Hook.

Use the following implementation pattern to achieve the above required Connector behavior: for each Entry found call Connector's "*addFindEntry(...)*" method. When finished, call "*getFindEntryCount(...)*" to get the number of Entries you have found – if it is 1, return the value returned by "*getFirstFindEntry(...)*" , otherwise return NULL.

For example: In a database Connector, "*modEntry(...)*" executes an **SQL UPDATE** query, using the Attributes of the entry parameter as database fields and the SearchCriteria in the search parameter to build the **WHERE** clause.

public void putEntry (Entry entry)

Required for AddOnly and Update modes. It is called once on each AssemblyLine iteration when the Connector is used in AddOnly mode, or for Update mode when no matching entry is found in the connected data source.

The goal of this method is to add/save/store the Entry object (passed in as parameter to this method) into the Connector's data source. So, a database Connector would execute an **INSERT SQL** statement using the Entry's Attributes' names and values and table fields names and values.

public void modEntry (Entry entry, SearchCriteria search, Entry old)

—or—

public void modEntry (Entry entry, SearchCriteria search)

Required for Update mode.

Before discussing the "*modEntry(...)*" method, a short clarification of the Update mode is necessary: When the AssemblyLine encounters a Connector in Update mode, it will first execute Connector's "*findEntry(...)*" method using the specified Link Criteria. If "*findEntry(...)*" finds no matching Entry, then the Connector's "*putEntry(...)*" method is called to add the Entry to the data source. If "*findEntry(...)*" finds exactly one Entry, the Connector's "*modEntry(...)*" method is called. Finally, if the "*findEntry(...)*" method finds more than one Entry, the "On Multiple Entries" hook is executed and depending on what the user specified either no Connector's calls are invoked or one of "*putEntry(...)*", alternatively "*modEntry(...)*" methods is invoked.

As seen above there are two variants of the "*modEntry(...)*" method – one with three and one with two input parameters. The two parameters that you get in both cases are: *entry*, the output mapped *conn* Entry, ready to be written to the data source; and *search*, the SearchCriteria to be used to make the modify call to the underlying system. When this method is invoked by the Update mode logic (the *"update(...)"* method of an AssemblyLineComponent), this will reference the actual SearchCriteria built from the Link Criteria (after evaluation of Attribute values, etc.).

The extra parameter is *old*. This is the original Entry in the data source as it looks right now, before the modification is applied. This information might be useful in certain cases like "rename" operations when you need the old name to perform the rename.

It is up to you to decide which of these methods to use. Of course you could implement both of them. One of them is sufficient for your Connector to support Update mode.

Following the analogy with the database Connector, "*modEntry(...)*" would execute an **SQL UPDATE** query, using the Attributes of the entry parameter as database fields and the data from the search parameter to build the **WHERE** clause of the SQL query.

public Entry queryReply (Entry entry)

Required for CallReply mode. It is called once on each AssemblyLine iteration when the Connector is used in CallReply mode.

This mode is appropriate when your Connector participates in some kind of request-response communication. The output mapped *entry* parameter contains the data necessary to perform the "call" or "request" part of the operation. For example, the Web Service Connector builds and transmits a SOAP call based on the Attributes in *entry*. The method then must build and return an Entry object from the reply/response data.

public void deleteEntry (Entry entry, SearchCriteria search)

Required for Delete mode.

Delete mode will cause the Connector to perform a "*findEntry(...)*" to try and locate the Entry to be deleted. If the "*findEntry(...)*" method returns exactly one Entry, the "*deleteEntry(...)*" method is called with this Entry and the Link Criteria used in the Lookup as parameters. If "*findEntry(...)*" returns zero or more than one Entries the corresponding Connector hooks are called. Depending on what the user specified in the script code, either nothing more is executed or the "*deleteEntry(...)*" method is called with the Entry specified by the user script via the AssemblyLineComponent method *setCurrent(entry)*. Unless the current entry is set in the On Multiple Found hook, nothing more happens, and control passes down the AssemblyLine.

Back to our database Connector example, "*deleteEntry()*" would execute an SQL DELETE statement.

public ConnectorInterface getNextClient()

The "*getNextClient()*" method is used for Connectors in Server mode to accept a client request. This method usually blocks until a client request arrives. When a request is received it creates and returns a new instance of itself. This new instance is then handed over to the AssemblyLine that spawns a new AssemblyLine thread for that Connector instance. The AssemblyLine then calls the "*getNextEntry()*" method on this new Connector instance in the new thread until there are no more Entries for processing. Right after the "*getNextClient()*" method returns and the AssemblyLine spawns a new thread to handle the client request, the AssemblyLine calls again "*getNextClient()*" to accept the next client request.

Since Connectors in Server mode handle client requests which require a response, the AssemblyLine will call the "*replyEntry(...)*" Connector method at the end of the AssemblyLine. Use this method to place your code that returns response to the client. In case your Connector might need to return multiple responses on a single request you can code the "*putEntry(...)*" method so that it returns an individual response Entry. In this case it will be the responsibility of the AssemblyLine developer to call the "*putEntry(...)*" method of the Connector by scripting and this fact has to be documented in the Connector's documentation.

When implementing a Connector in Server mode, you also have to take care about terminating the Connector on external request. Place your termination code in the "*terminateServer()*" method. Take into consideration that this method can be called on the master Connector instance that accepts client requests and also on a child Connector instance processing a client request. In both cases proper termination should happen: it is usually a good termination practice to stop accepting new requests from the master Server Connector instance but let all child Connectors finish their processing. The "*terminateServer()*" method usually sets some flag that is checked by the "*getNextClient()*" method of the master Server Connector instance – if termination is requested the "*getNextClient()*" method will return NULL. This is a signal to the AssemblyLine that this Server Connector has terminated and the AssemblyLine will not call anymore its "*getNextClient()*" method.

public void terminate ()

The "*terminate(...)*" method is called by the AssemblyLine after it has finished cycling and before it terminates. You would put here any cleanup code, that is, release connections, resources that you created in the "*initialize(...)*" method or later during processing.

The methods listed above are the core *ConnectorInterface* methods that bring life to your Connector. And remember, you only need to implement the methods corresponding to the Connector modes that your Connector will support.

Modes to methods mapping:

When you write a connector you should take into account that users may call the connector methods in no particular order. This means you should have sanity checks on each method in case the connector requires certain methods to be called before others. From a Tivoli Directory Integrator server perspective the methods called on a connector is determined by the value of the mode parameter. In this section you will see the call order for methods for each mode. When the AssemblyLine uses a connector it always calls `initialize()` as the first method before any other methods are called. However, it is possible that the user sneaks in a call to other methods before this is called.

Table 85.

Mode	Methods	Comments

Iterator	`Initialize()`
	`selectEntries()`
	`getNextEntry()`
	`terminate()`
		getNextEntry should return NULL to signal end of input.

Table 85. (continued)

Mode	Methods	Comments

AddOnly	`Initialize()`
	`putEntry()`
	`terminate()`	

Lookup	`nitialize()`
	`findEntry()`
	`terminate()`
		If you find more than one entry you should use `clearFindEntries()` and `addFindEntry()` for each entry found in this method.

Delete	`Initialize()`
	`findEntry()`
	`deleteEntry()`
	`terminate()`	

Update	`Initialize()`
	`findEntry()`
	`putEntry()`
	`modEntry()`
	`terminate()`
		If findEntry returns a single entry, `modEntry` will be called. If findEntry returns null, `putEntry` will be called.

Delta	`Initialize()`
	`findEntry()`
	`putEntry()`
	`modEntry()`
	`deleteEntry()`
	`terminate()`
		See note below.

How to implement a Delta mode Connector:
First of all, to enable delta mode for your connector you must add "Delta" to the list of supported modes. Delta mode is a bit special since it can be emulated by the AssemblyLine or directly implemented by the connector. Emulated delta mode simply means that the incremental update is generated by the AssemblyLine based on what it is returned by the `findEntry()` method and what is being written to the target system. If the target system supports incremental updates you can code your connector by translating a delta Entry object to the underlying protocol of your connector. In the latter case you configure your connector by returning true in the `isDeltaSupported()` method. This will cause the AssemblyLine to forward the delta Entry directly to your connector's `putEntry()`, `modEntry()` or `deleteEntry()` methods, bypassing `findEntry()` and the algorithm to compute the delta entry.
Query Schema behavior:
In Tivoli Directory Integrator 7.1 a default behavior for schema discovery is implemented for all Connectors. This default behavior is used by Connectors that do not implement their own logic of schema processing, that is, do not override the `querySchema(Object)` method. The default behavior depends on the Parsers that a Connector has (if any).

Static Schema

Each Connector static schema is determined by its own static schema plus the static schema of the Parser it uses. Both schemas are displayed in the Connector Input/Output Map.

Under a static schema we understand the schema that is configured in the `tdi.xml` file for the Connectors and Parsers. To add a static schema definition to your Connector, Parser or Function definition file, add a <Schema> tag inside the <Connector>, <Parser> or <Function> element. The name of the Schema should be "input" or "Output".

For example:

```
<Connector name="ibmdi.Mailbox">
 <Schema name="Input">
  <SchemaItem>
   <Name>mail.body</Name>
   <Syntax>javax.mail.Multipart</Syntax>
  </SchemaItem>
 </Schema>
 <Schema name="Output">
```

```
  <SchemaItem>
   <Name>Flag.Answered</Name>
   <Syntax>boolean</Syntax>
  </SchemaItem>
 </Schema>
</Connector>
```

Dynamic Schema

This type of schema will require an interaction with the System to which the Connector is configured to connect. To do so the Connector may ask the configured Parser if it could discover a schema by using its configuration.

Implementing querySchema():

ConnectorInterface also provides other methods that address aspects of the possible use of a Connector and which you might want to implement. One example is "*querySchema(...)*". This method returns the schema of the connected data source . If you implement it, the Config Editor presents the returned values as the Connector's Schema.

These return values are stored as a Vector of Entry objects, one for each column/attribute in the schema. For example, a database Connector would return one Entry for each column in the connected database table.

Each Entry in the Vector returned should contain the following attributes:

name	The name of the attribute (column, field, etc.) Required.
syntax	The syntax (like **VARCHAR** or **TIMESTAMP**) or expected value type of this attribute. Optional

Specified by: *querySchema* in ConnectorInterface

Parameters: *source* - The object on which to discover schema. Usually NULL. This may be an Entry or a string value.

Returns: A vector of *com.ibm.di.entry.Entry* objects describing each entity, or in the case of error, a java.lang.Exception is thrown.

Using a Parser in your Connector

If your connector extends the base implementation of the TDI connector (com.ibm.di.connectors.Connector), you can invoke the *initParser()* method to initialize the associated parser:

```
/**
 * Initialize the connector's parser with input and output streams. If the parser
 * has not been loaded then an attempt is made to load it. The input and output objects
 * may be Stream objects (InputStream,OutputStream), java.io.Reader object, String object,
 * java.net.Socket, byte and character array objects.
 *
 * @param  is  The input object.
 * @param  os  the output object.
 * @exception  Any exception thrown by the parser
 * @see    #getParser
 */
public void initParser (Object is, Object os) throws Exception;
```

You have to provide the input and/or output streams the parser will use for its read/write operations. The mode of your connector typically determines which way the flow goes (note that your *initialize(Object obj)* connector method will have the connector mode in the "obj" object). You are not required to initialize the parser at the time of connector initialization, but you should do so unless there is a good reason to

initialize it elsewhere. In any case you should invoke the *initParser()* method to properly initialize the parser with logging objects, debug flags and other standard TDI objects/behaviors.

The parser can be chosen either by the user or you can hide the parser selection and either provide the configuration in your "tdi.xml" file or programmatically configure the parser in your connector (or both).

1. **Let the user choose the parser.**

   In this case you must set the parameter "parserOption" in your connector's "`tdi.xml`" file to the value "true". Once this field is defined the selection of the parser is delegated to the user through a standard user interface (note that you can prefill the parserConfig section of your `tdi.xml` file with a default parser). Here is a snippet from the FileSystem connector's "tdi.xml" file showing "parserOption" as "Required", which means the connector requires a parser (that is, an error is thrown if none is defined in the configuration):

   ```
   <Connector name="ibmdi.FileSystem">
    <Configuration>
    ...
     <parameter name="parserOption">Required</parameter>
    </Configuration>
   </Connector>
   ```

   The value for the "parserOption" parameter can be "Required", "Useless" (no parser allowed) or "Optional".

2. **Use a predefined parser using the "tdi.xml" file.**

   You can include the parserConfig section in your "tdi.xml" file if you always use the same parser, for example if you inherit from the CSV Parser:

   ```
   <Connector name="myconnector">
    <Configuration>
     <parameter name="parserOption">Required</parameter>
    </Configuration>
    <Parser>
     <InheritFrom>system:/Parsers/ibmdi.CSV</InheritFrom>
     ... Optional parameter values to make the parser functional
    </Parser>
   </Connector>
   ```

3. **Configure the parser at runtime.**

   Your connector has access to the ConnectorConfig object via the *Connector.getConfiguration()* method. Through the ConnectorConfig object you can obtain the ParserConfig interface object for the connector. Use that object to configure the parser before you invoke the *initParser()* method:

   ```
   import com.ibm.di.config.interfaces.ConnectorConfig;

    public void initialize(Object obj) throws Exception {

     // Check mode
     String mode = "" + obj;
     boolean isIterator = mode.equals(ConnectorConfig.ITERATOR_MODE);

     ConnectorConfig cc = (ConnectorConfig)getConfiguration();

     // Get the parser config object
     ParserConfig parser = cc.getParserConfig();

     // -- use the csv parser and set the column separator parameter
     parser.setParameter("parserType", "com.ibm.di.parser.CSVParser");
     parser.setParameter("csvColumnSeparator", "\t");

     if(isIterator)
      initParser(inputStream, null);
     else
      initParser(null, outputStream);
   ```

Once the parser has been initialized you can invoke the *readEntry()* and *writeEntry()* methods to translate com.ibm.di.entry.Entry objects to and from the stream format defined by the parser. You typically invoke the *readEntry()* method in your *getNextEntry()* method and the writeEntry method from your putEntry method. You obtain the parser interface handle through the *getParser()* method.

4. **Optional parser and dynamic reinitialization**

   If your connector can function with or without a parser you can invoke the *hasParser()* method to determine whether a parser is configured or not:

   ```
   if(hasParser())
      doSomething();
   ```

   If you use multiple instances of the parser during the life time of your connector you should close the parser interface to ensure data is written to the outputstream and that system resources are released. The methods used to re-initialize a parser can differ based on which parser you use but the following method calls should be sufficient for most parsers:

   ```
   // Close parser to release system resources
   if(getParser() != null)
    getParser().closeParser();

   // assuming you just got a new input stream ... reinitialize the parser
   initParser(inputStream, null);
   ```

   When your connector is terminated it will automatically invoke the *closeParser()* method if one is in use by the connector.

Logging from a Connector

The **com.ibm.di.connector.Connector** class that your Connector will be extending, has a number of methods to enable you to log messages to the AssemblyLine's configured log files. The simplest way of logging is using one of the following methods:

```
/**
  * Log a message to the connector's log. The message is prefixed by the connector's
  * name.
  *
  * @param  msg  The message to write to the log
  */
 public void logmsg(String msg)

 /**
  * Log a debug message to the connector's log
  *
  * @param  msg  The message to write to the log
  */
 public void debug(String msg)
```

You can call these methods with code like

```
logmsg("initializing my connector");
```

This will cause your string to be issued to the AssemblyLine's configured log appenders, at INFO level. If you want to do more advanced logging, the com.ibm.di.connector.Connector class also has this field:

```
/**
  * The log object for logging messages
  */
  protected  com.ibm.di.server.Log        myLog;
```

This **com.ibm.di.server.Log** class has many methods for logging. You could therefore use the myLog object to do logging like this:

```
myLog.logerror("Something very bad happened");
```

This issues a message to the log(s) at ERROR level. There are corresponding methods for logging at different levels, like loginfo() and logfatal().

Building the Connector's source code

When building the source code of your Connector, set up your `CLASSPATH` to include the jar files from the "`jars/common`" folder of the IBM Tivoli Directory Integrator installation. At minimum you would need to include "`miserver.jar`" and "`miconfig.jar`".

Note: When integrating your Java code with IBM Tivoli Directory Integrator, pay attention to the collection of pre-existing components that comprise IBM Tivoli Directory Integrator, notably in the *jars* directory. If your code relies upon one of your own library components that overlap or clash with one or more that are part of the Tivoli Directory Integrator installation there will most likely be loader problems during execution.

Implementing the Connector's GUI configuration form

Introduction

When you create a custom Tivoli Directory Integrator component you also have to provide an additional file that describes your component to TDI. This file is located at the root of your jar file and is named *tdi.xml*. The syntax and contents of this file is described in this document.

The first part of this section explains the format of the tdi.xml file and also shows the minimum requirements for a component definition file.

The second part of this section focuses on the form definition and the various options you have when you define a form. This form definition is used by the Tivoli Directory Integrator Configuration Editor to let the user configure your component. While the UI options in the form definition are basic and somewhat limited, you can still perform advanced operations using your own custom java based UI components as well as associating scripts with form events.

tdi.xml file format

The files are created in XML format looking much the same as a Tivoli Directory Integrator Configuration file.

A skeleton for the file could look something like this:

```
<?xml version="1.0" encoding="UTF-8">
<MetamergeConfig version="7.0">
 <Folder name="Connectors">
  <Connector name="CustomConnector">
   <Configuration>
    <parameter name="connectorType">com.acme.CustomConnector</parameter>
    ... more parameters ...
   </Configuration>
  </Connector>
 </Folder>
 <Folder name="Forms">
  <Form name="com.acme.CustomConnector">
   <TranslationFile>CustomConnector</TranslationFile>
   ... many more elements which will be defined later ...
  </Form>
 </Folder>
</MetamergeConfig>
```

This defines a Connector named system:/Connectors/CustomConnector. The Java class that implements this Connector is `com.acme.CustumerConnector.class`.

Localization of labels and descriptions in this file can be provided by adding properties files with the *locale* identifier in the standard way. In this example, the properties file is CustomConnector.properties.

Then the German version of this file would be CustomConnecter_de.properties, and the Brazilian Portuguese version would be CustomConnector_pt_BR.properties. The individual properties in these localized files take the same keys, but with localised values. Each line is of the format

```
key=value
```

Comments in these files can be included by starting the line with a # (hash).

Basic Component Definitions: When you first create your component definition file you add the main sections for the components your jar file contains. For each component you add a section where you as a minimum define Java class. The syntax is as follows for the three main components:

Table 86.

Component Type	Minimum Section Contents

Connector	`<Folder name="Connectors">`
	`<Connector name="`*your_name*`">`
	`<Configuration>`
	`<parameter name="connectorType">`*your_javaclass_name*`</parameter>`
	`</Configuration>`
	`</Connector>`
	`</Folder>`

Parser	`<Folder name="Parsers">`
	`<Parser name="`*your_name*`">`
	`<parameter name="class">`*your_javaclass_name*`</parameter>`
	`</Parser>`
	`</Folder>`

Function	`<Folder name="Functions">`
	`<Function name="`*your_name*`">`
	`<Configuration>`
	`<parameter name="javaclass">`*your_javaclass_name*`</parameter>`
	`</Configuration>`
	`</Function>`
	`</Folder>`

In addition you should always include a form definition for each of your components. This is to prevent the configuration editor to report errors of missing forms. If your component has no configurable parameters you should include a form that says so.

Note: The current configuration object that the *Form* refers to is always the *connectorConfig/parserConfig/functionConfig* object. If you need to access the main component's parameters you should use the "*config.getParent()*" method to obtain for example the ConnectorConfig interface for the configuration.

Install Location: When you start either the configuration editor or the server there is a component called the Tivoli Directory Integrator Loader that runs through its configured jar directories looking for *.jar/*.zip files that contain an "tdi.xml" file at its root level. All the definitions in these files are put into the system namespace.

The locations of these files are:

- *TDI_Install_directory*`/jars` and any subdirectory therein
- Any files/directories specified by the `com.ibm.di.loader.userjars` property (`etc/global.properties`)

When you put your jar file in either of these directories your component will show up in the configuration editor with the name you chose as part of the system namespace.

Note: Adding your jar file to the CLASSPATH or PATH alone does not include it in the system templates and hence will not be visible to the user.

Form description

The form description is used to provide custom input panels for components. While most of the user interface in the configuration editor is static, most components need specific user interfaces to let the user define its behavior.

Component/Form Association: The form definition defines the input fields and labels that the configuration editor will build when you open the configuration for a component. The binding between the component (for example, connector, parser) and its form is through the Java class of the component. Using the example above, the connectorConfig has a "connectorType" parameter that defines the implementing class for the component (com.acme.CustomConnector). When a component of this type is presented to the user, the configuration editor will look for a form with the same name as the implementing Java class.

Form/Configuration Binding: When the form has been created it also has a binding object for each parameter to the configuration object. These binding objects will set the initial value of the input field (using the default value provided by the form if the configuration object returns null for the value) and also function as the controller between the input field and the configuration object. When the input field changes its value the binding will update the configuration object and vice versa. The configuration object is read and updated using the primitives of the configuration object (for example, BaseConfiguration.getParameter/setParameter). It is possible to have the binding object invoke specific methods rather than using the primitives, but for component developers this is rarely needed.

Form Definition: Forms are defined the same way as components are defined. Below is an example of a form with three input fields and one event handler trapping changes to one of the parameters. The form definition is divided into two sections; General and Advanced. The General section contains two parameters ("firstParameter" and "$GLOBAL.debug"), whereas the second section contains just one parameter ("secondParameter").

We have only defined a label for the two parameters; $GLOBAL.debug is a Tivoli Directory Integrator global parameter that enables detailed logging when checked.

```
<Folder name="Forms">
 <Form name="com.acme.CustomConnector">
  <TranslationFile>CustomConnector</TranslationFile>
  <parameter name="title">title_key</parameter>
  <parameter name="formevents">function firstParameter_changed() { form.alert("First param modified"); }</parameter>
  <FormSectionNames>
   <ListItem>General</ListItem>
   <ListItem>Advanced</ListItem>
  </FormSectionNames>
  <FormSection name="General">
   <FormSectionNames>
    <ListItem>firstParameter</ListItem>
    <ListItem>$GLOBAL.debug</ListItem>
   </FormSectionNames>
  </FormSection>
  <Formsection name="Advanced">
   <parameter name="title">Advanced_Title</parameter>
   <parameter name="initiallyExpanded">false</parameter>
   <FormSectionNames>
    <ListItem>secondParameter</ListItem>
   </FormSectionNames>
  </FormSection>
  <FormItem name="firstParameter">
   <parameter name="label">first_param_label</parameter>
  </FormItem>
  <FormItem name="secondParameter">
   <parameter name="label">second_param_label</parameter>
  </FormItem>
 </Form>
</Folder>
```

The translation file (`CustomConnector_en.properties`) would contain:

```
title_key=This is the title/heading that appears at the top of the form
Advanced_Title=This is the title heading for the section for Advanced Users
first_param_label=First Param Label
second_param_label=Second Param Label
```

Forms definition elements: A FormSection element contains a list of FormSections or FormItems. This list has the tag <FormSectionNames>; the FormSection can optionally include a title and redefinitions of FormItems. These FormItems inherit from the FormItem in the Form that the FormSection is part of. This allows you to, for example, override the Tooltip for that FormItem. The Form contains a list of FormSections; this list is tagged <FormSectionNames>. In any list of FormSections, the word $Mode will be replaced by the current mode for the Connector. This allows you to show parameters depending on the mode of the Connector.

Here is a somewhat complex example of a complete form:

```
<Form name="com.ibm.di.connector.FileConnector">
   <TranslationFile>NLS/idi_conn_filesys</TranslationFile>
   <FormItemNames>
    <ListItem>filePath</ListItem>
    <ListItem>fileAwaitDataTimeout</ListItem>
    <ListItem>fileAppend</ListItem>
    <ListItem>exclusiveLock</ListItem>
    <ListItem>$GLOBAL.debug</ListItem>
    <ListItem>$GLOBAL.help</ListItem>
   </FormItemNames>
   <FormSectionNames>
    <ListItem>$Mode-General</ListItem>
    <ListItem>$Mode-Advanced</ListItem>
   </FormSectionNames>
   <FormSection name="Iterator-General">
    <FormSectionNames>
     <ListItem>filePath</ListItem>
    </FormSectionNames>
    <FormItem name="filePath">
     <parameter name="description">path_desc_in</parameter>
    </FormItem>
    <parameter name="title">General_title</parameter>
   </FormSection>
   <FormSection name="AddOnly-General">
    <FormSectionNames>
     <ListItem>filePath</ListItem>
     <ListItem>fileAppend</ListItem>
    </FormSectionNames>
    <FormItem name="filePath">
     <parameter name="description">path_desc_out</parameter>
    </FormItem>
    <parameter name="title">General_title</parameter>
   </FormSection>
   <FormSection name="Iterator-Advanced">
    <FormSectionNames>
     <ListItem>fileAwaitDataTimeout</ListItem>
     <ListItem>exclusiveLock</ListItem>
    </FormSectionNames>
    <FormItem name="exclusiveLock">
     <parameter name="description">exlock_desc_in</parameter>
    </FormItem>
   </FormSection>
   <FormSection name="AddOnly-Advanced">
    <FormSectionNames>
     <ListItem>exclusiveLock</ListItem>
    </FormSectionNames>
    <FormItem name="exclusiveLock">
     <parameter name="description">exlock_desc_out</parameter>
    </FormItem>
   </FormSection>
   <FormItem name="exclusiveLock">
    <parameter name="label">exlock_label</parameter>
    <parameter name="description">exlock_desc</parameter>
    <parameter name="syntax">boolean</parameter>
   </FormItem>
   <FormItem name="fileAppend">
```

```
 <parameter name="description">append_desc</parameter>
 <parameter name="label">append_label</parameter>
 <parameter name="syntax">boolean</parameter>
</FormItem>
<FormItem name="fileAwaitDataTimeout">
 <Values>
  <ListItem>-1</ListItem>
  <ListItem>10</ListItem>
  <ListItem>60</ListItem>
 </Values>
 <parameter name="description">time_desc</parameter>
 <parameter name="label">time_label</parameter>
 <parameter name="syntax">DROPEDIT</parameter>
</FormItem>
<FormItem name="filePath">
 <Values>
  <ListItem>&lt;&gt;</ListItem>
 </Values>
  <parameter name="description">path_desc</parameter>
  <parameter name="label">path_label</parameter>
  <parameter name="script">selectFile</parameter>
  <parameter name="scriptLabel">path_sript_label</parameter>
  <parameter name="scripthelp">path_script_help</parameter>
  <parameter name="syntax">DROPEDIT</parameter>
</FormItem>
<parameter name="title">CONN_TITLE</parameter>
</Form>
```

Definition of XML Tags:

- <Form> defines a Form.
- Inside a <Form> there may be the following tags:
  - <TranslationFile> - Defines the name of the translation file.
  - <FormItemNames> - A list of FormItems for use with the old Config Editor, which does not understand FormSections.
  - <FormSectionNames> - The list of FormSections/FormItems to show.
  - <FormSection> - Defines a FormSection.
  - <FormItem> - Defines a FormItem.
  - <parameter> - Some possible parameters are:
    - title - The Title of the form. It will be translated if there is a translation file, and the key is found. If there is no translation file at all, or the key is not found, the value itself will be used.
    - description - More description of the form.
    - formscript - Javascript code to be executed every time a button with a script is pushed, for example, to define methods the script for the button can use.
    - formevents - This parameter is also JavaScript code, to be executed when the form is created. Its purpose is to define methods that can react to fields being set to certain values, for example to populate dropdowns for other fields. If a field has the name fieldName, and the method fieldName_changed() has been defined by this script, then that method will be called when fieldName changes value. The formevents parameter can be a long CDATA section in the XML file.
- Inside a <FormSection> there may be the following tags:
  - <FormSectionNames> - As in a Form.
  - <FormItem> - As in a Form, but with implicit inheritance from any similar named FormItem in the enclosing Form.
  - <parameter> - The following parameters are recognized:
    - title - A title causes the FormSection to be displayed different.
    - description - more description for the FormSection.

- initiallyExpanded - If this parameter is set to false, then the FormSection will initially not be expanded. The default value is true.

- Inside a <FormItem> there may be the following tags:
  - <Values> - Specifies a list of values for a droplist/dropedit.
  - <LocalizedValues> - Map from values in <Values> to keys that will be looked up in the translation file.
  - <parameter> - Defines a parameter for the FormItem, as per the table below.
- <ListItem> defines an item in a list.
- Inside <LocalizedValues> there will be the following tags:
  - <Item> - One item in the Map.
- Inside <Item> there will be the following tags:
  - <Key> - The key.
  - <Value> - The value.
- Example for <LocalizedValues>:

```
<LocalizedValues>
 <Item>
  <Key>After every database operation</Key>
  <Value>Localized.After.every.database.operation</Value>
 </Item>
 <Item>
  <Key>After every database operation (Including Select)</Key>
  <Value>Localized.After.every.database.operation.Including.Select</Value>
 </Item>
 <Item>
  <Key>Manual</Key>
  <Value>Localized.Manual</Value>
 </Item>
 <Item>
  <Key>On Connector close</Key>
  <Value>Localized.On.Connector.close</Value>
 </Item>
</LocalizedValues>
```

XML Translation considerations: Instead of merging the translated values from the properties file into the XML file, there is a new tag in the Form, <TranslationFile>. The correct local version of this translation file will be read in when using the Config Editor, and the values will then be used.

Example for the File Connector: the tdi.xml file for the File Connector contains this tag for the Form:

```
<TranslationFile>NLS/idi_conn_filesys</TranslationFile>
```

You would package this XML file with all the NLS/idi_conn_filesys.properties files in the jar file.

Parameter Definitions: These are the recognized parameters that can be used in a FormItem.

Table 87. FormItem parameters

Keyword	Description
label	The label appearing in the left column of the form (for example, LDAP URL)
description	The tooltip for the parameter
default	Default value for the parameter. The preferred way of providing a default value is in the component configuration itself (in the tdi.xml file). This default value will only be set if the user uses the CE to view/modify the configuration for the component.
script script2	Specifying this parameter adds a button to the right of the input field. When the button is clicked, the named JavaScript function is executed. *Script2* allows for a second button to the right of the first one.

Table 87. FormItem parameters (continued)

Keyword	Description
scriptLabel scriptLabel2	The button text
scriptHelp scriptHelp2	Tooltip for the script button
syntax	Specifies the syntax of the parameter. This also affects the choice of UI control used to represent the value. See the syntax section for more info.
reflect	If present the binding will use this method to get/set the parameter value. The binding will prepend "get" or "set" accordingly to this value (for example, specify Name to invoke getName and setName). This is only used when the configuration object performs specific logic when getting/setting a parameter value. For component developers this is rarely needed as component configurations only have get/set primitives.

Dynamic Values: The *values* list can contain static and dynamic values. The dynamic values are expanded and added to the array at runtime to populate the dropdown list.

Table 88.

Value	Description
@ASSEMBLYLINES@	Adds all known AssemblyLines to the array
@CONNECTORS@	Adds all known connectors to the array
@PARSERS@	Adds all known parsers to the array
@FUNCTIONS@	Adds all known function components to the array
@ATTRS@	Adds all attributes from the input map

Syntax: The syntax parameter for a FormItem can have any of the following values:

Table 89.

Value	Description
String	This is the default syntax. A one line text field is created for text input.
Password	A password field is created for text input. Be aware that if the user has configured a password store then FormUI will not insert the value in the configuration object but insert a property reference. The actual value is then stored in the password store. If you modify this parameter via script or java code make sure to invoke *BaseConfiguration.setProtectedParameter()* instead of *BaseConfiguration.setParameter()*. The setProtectedParameter will automatically create a new property if there isn't one in place already. If the password store is not configured setProtectedParameter will simply invoke setParameter instead.
Boolean	A checkbox is created for true/false values
Droplist Dropedit	Dropdown with values from the values parameter. Dropedit is the editable version where the user also has a text field to specify a custom value. See Dynamic Values for special values.
TextArea	Creates a text area control for multi-line text input
Script	Creates a button that invokes a script
Static	Creates a text label for viewing only (same as TextArea, but readonly)

Table 89. (continued)

Value	Description
EditorWindow	This syntax causes the form to be a tabbed pane. The non-editorwindow parameters appear in the left most tab whereas each editorwindow parameter has its own tab with an editor input control. Used when you need the complete display area for input (for example, scripts)

Component	This enables you to provide your own UI component if you need complex input mechanisms or otherwise want more control over the UI. Specify the java class name in the component keyword that you want inserted into the form:
	
	syntax:component
	component:pub.test.CustomUI
	
	Version 7.x – Eclipse SWT Components:
	
	The class is instantiated by FormWidget2 at runtime and should be an SWT Control subclass (something that can be a child of a Composite). Also, it must have a constructor as shown in this example:
	`package pub.test;`
	
	`import org.eclipse.swt.widgets.Composite;`
	`import com.ibm.tdi.eclipse.widgets.FormWidget2;`
	`import com.ibm.di.config.interfaces.BaseConfiguration;`
	
	`public class CustomUI extends Composite {`
	`/*`
	`* form - the FormWidget2 object`
	`* parent – The Composite in which this control is placed`
	`* config - the config object being edited`
	`* paramname - the parameter of config being edited`
	`*/`
	`public CustomUI(FormWidget2 form, Composite parent, BaseConfiguration config,`
	`String paramname) {`
	`super(parent, 0);`
	`}`
	
	`}`
	
	This component will be placed in a Composite using GridLayout. Do not set the GridData of the custom UI object as this is done by the form widget after creating the custom class.

Form Scripts: In your form definitions you can add calls to script functions. These functions execute in the form's script engine. The form's script engine provides the following predefined objects:

- **form** - Represents the com.ibm.tdi.eclipse.widget.FormWidget2 instance managing this form.
- **config** - A handle to the configuration objects this form operates on (for example, the connection configuration for a connector, parser config for parser etc).
- **attributeName** - The name of the FormItem.
- **system** - An instance of the com.ibm.di.function.UserFunctions class.

Examples

Look at TDI's components in the configuration editor to find an example you find suitable. Use a zip/jar tool (for example, winzip, unzip) and extract the "tdi.xml" file from the component's jar file (*TDI_install_dir*`/jars/components` subdirectory).

Also, the *examples/connector_java* folder of this package contains the "`tdi.xml`" file of the Directory Connector.

Connector Reconnect Rules definition

In order to take advantage of the Tivoli Directory Integrator Reconnect feature, the Connector's .xml file may contain rules that tailor the Connector's response to interruptions in connectivity. These rules are in addition to any built-in rules of the Reconnect engine of the Tivoli Directory Integrator Server.[1]

The rules for the particular Connector appear in the "connectors" section as a sibling of the "connectorConfig" sub-section like this:

```
<Connector name="CustomConnector">
 <Configuration>
   ... various configuration options ...
 </Configuration>
 <Reconnect>
  <ReconnectRules>
   <Rule>
    <parameter name="exceptionClass">java.sql.SQLException</parameter>
    <parameter name="exceptionMessageRegExp">^I/O.*</parameter>
    <parameter name="action">reconnect</parameter>
   </Rule>
   <Rule>
    <parameter name="exceptionClass">java.sql.SQLException</parameter>
    <parameter name="exceptionMessageRegExp">^Io.*</parameter>
    <parameter name="action">reconnect</parameter>
   </Rule>
   ... more rules go here ...
  </ReconnectRules>
 </Reconnect>
```

Each rule has the following parameters:

```
exceptionClass: fully qualified name of the Java class of the exception
exceptionMessageRegExp: regular expression in Java syntax
action: error or reconnect
```

Parameters "exceptionClass" and "exceptionMessageRegExp" are optional – if not specified, the rule will match all exception classes and all exception messages respectively.

For a detailed description of the regular expression syntax used in "exceptionMessageRegExp", please see the the JavaDoc of the `java.util.regex.Pattern` class at http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html.

Example:

```
<Connector name="ibmdi.ReconnectTest">
 <Configuration>
  <parameter name="connectorType">com.ibm.di.connector.ReconnectTestConnector</parameter>
   </Configuration>
 <Reconnect>
  <ReconnectRules>
   <Rule>
    <parameter name="exceptionClass">java.io.IOException</parameter>
    <parameter name="exceptionMessageRegExp">.*file not found.*</parameter>
    <parameter name="action">error</parameter>
   </Rule>
   <Rule>
    <parameter name="action">reconnect</parameter>
   </Rule>
  </ReconnectRules>
 </Reconnect>
</Connector>
```

1. In order for Tivoli Directory Integrator to remain backwards-compatible, there are two built-in rules that emulate previous version's behavior. Unless reconnect is switched off entirely for a Component, a reconnect will be attempted for all Exceptions of type `java.io.IOException` and `javax.naming.CommunicationException`.

Packaging and deploying the Connector

Now that we have the Connector source code compiled and supplied the "`tdi.xml`" file, we are ready to package and deploy the Connector.

What you need to do is create a jar file (typically with the same name as that of the Connector) and include in it:

1. The class file(s) of the Connector
2. The "`tdi.xml`" file, in the root of the jar file. If you are using translated files, also include them using the standard Java internationalization schema, for example "CustomConnector_de.properties" for German, "CustomConnector_fr.properties" for French, "CustomConnector_pt_BR.properties" for Brazilian Portuguese.

After you have created the jar file of the new Connector, you need only drop that jar file in the "`jars/connectors`" folder of the IBM Tivoli Directory Integrator installation. The next time the system starts up, it will automatically load the new Connector and make it ready for use.

Developing a Function Component

Implementing a Function Component (FC) follows very much the pattern of developing a Connector. A Function Component is actually easier to implement because of fewer dependencies on the AssemblyLine workflow.

Implementing Function Component Java source code

Similar to the Connector foundation classes, we have "*com.ibm.di.fc.FunctionInterface*" and the "*com.ibm.di.fc.Function*" abstract class that implements the interface (the Java sources of both classes are included in the "`fc`" folder of this package).

You will usually implement your FCs by subclassing the "*com.ibm.di.fc.Function*" class. These are the most important methods you will usually need to implement:

public void initialize (Object obj)

Put any initialization code here – reading FC's parameters, allocating resources, etc. When the FC is placed into an AssemblyLine, the AssemblyLine will call the "*initialize(...)*" method once, on startup.

When the FC is created and used programmatically the "*initialize(...)*" method must be called right after constructing the FC object and setting its parameters, and before calling its "*perform(...)*" method.

public Object perform (Object obj)

The "*perform(...)*" method is the actual implementation of the business logic of your FC. In contrast to the Connector where you have different Connector modes and different methods to implement for each of them (*getNextEntry()*, *findEntry()*, etc.) all that a FC is supposed to do is implemented in the "*perform(...)*" method.

The general contract for the "*perform(...)*" method is that it receives some data on input and based on that input it produces some output data. There are no other assumptions. As you will see below it is not even necessary that your FC works with Entry objects.

When the FC is placed into an AssemblyLine, the AssemblyLine calls its "*perform(...)*" method on each iteration. In the AssemblyLine context the "*perform(...)*" method will be given an Entry object as input parameter (this is the Entry object constructed by the Output Attribute Mapping process - also called the *conn* Entry). And it is supposed to return an Entry object as well; the AssemblyLine will feed the returned Entry object to the Input Attribute Mapping process, the result of which is applied to the AssemblyLine's work Entry - in other words, the returned Entry is the *conn* Entry in the Input Map. If you want to enable your FC to be placed into an AssemblyLine, you need to support this "Entry on input – Entry on output" behavior.

You might also code the "*perform(...)*" method so that it receives non-Entry objects on input and returns non-Entry objects on output. This could facilitate the process of programmatically creating and calling an FC. No Attribute Mapping will be done if you use this method.

public void terminate()

The FC's "*terminate(...)*" method is called by the AssemblyLine after it has finished cycling and before it terminates. You would put here any cleanup code, that is, release connections, resources that you created in the "*initialize(...)*" method or later during processing.

When using an FC programmatically, you must call the "*terminate(...)*" method after you have finished using that FC instance.

Building the Function Component source code

When building the source code of your Function Component, include in your `CLASSPATH` the jar files from the "`jars`" folder of the IBM Tivoli Directory Integrator installation. As a minimum, you would need to include "`miserver.jar`" and "`miconfig.jar`".

Note: When integrating your Java code with IBM Tivoli Directory Integrator, pay attention to the collection of pre-existing components that comprise IBM Tivoli Directory Integrator, notably in the *jars* directory. If your code relies upon one of your own library components that overlap or clash with one or more that are part of the Tivoli Directory Integrator installation there will most likely be loader problems during execution.

Implementing the Function Component GUI configuration form

The FC GUI is implemented in the same way as for a Connector. Use a "tdi.xml" file to describe the FC configuration form by using the same syntax as used for Connectors.

Packaging and deploying the Function Component

Packaging and deploying an FC is just like packaging and deploying a Connector:

You need a jar file that contains:

1. The class file(s) of the Function Component
2. "`tdi.xml`" file placed in the root of the jar file (and optionally "`MyFunction_??.properties`" files if you want to support different languages)

After you have created the jar file of the new Function Component, you only need to drop that jar file in the "`jars/functions`" folder in the IBM Tivoli Directory Integrator installation. The next time the IBM Tivoli Directory Integrator is started it will automatically load the new Function Component and it will be ready for use.

Developing a Parser

Parsers are used in conjunction with a transport Connector to interpret or generate the content that travels over the Connector's byte stream. However, sometimes you may want to parse data that is presented in a very specific format; for this purpose you will need to implement your own Parser.

Implementing the Parser Java source code

All Tivoli Directory Integrator Parsers implement the `com.ibm.di.parser.ParserInterface` Java interface. This interface provides a number of methods to implement that are common to all parsers. Usually the parsers that you write will not require implementing all methods provided by the interface but only a subset of them. For this purpose you can use the `com.ibm.di.parser.ParserImpl` abstract class that implements the `ParserInterface`. The `ParserImpl` class contains the core Parser functionality so you can subclass it when implementing your own Parser.

There are two types of parsers: ones that read from a stream and return an Entry; and others that take an Entry and write it to a stream.

Once the Parser is constructed you have to configure it. This includes setting the input/output streams and configuring some additional parameters if needed. This is usually made by the hosting component (for example, a Connector). When finished with this job, the next step is initialization of the Parser where resources for future needs are allocated and any other initialization takes place. Generally the hosting component takes care of both configuring and calling the initialization method of the Parser. Next comes the most significant moment of using the Parser – writing or reading the entries. This is the place where the actual parsing happens. Finally, when the Connector has finished transporting the entries, the Parser must be closed. When closed, the Parser releases the resources that were used in the previous stages as well as closing the input and output streams.

For an example of a Parser implementation, look at the `ExampleParser.java` Parser included in Tivoli Directory Integrator. These are some of the important methods you will usually need to implement:

public void setInputStream(InputStream is)
: Set the input stream attribute of the Parser here. This method is overloaded and can take "String" and "Reader" as arguments, too. The abstract class `com.ibm.di.parser.ParserImpl` provides implementations for the three methods that set the input stream of the Parser. If you subclass `com.ibm.di.parser.ParserImpl` you can override some of these methods or leave the default implementation of the super class. However, if you implement the interface you will have to provide implementation for all of them.

  Note that when you open the input stream, it is your responsibility to close it. This is usually done in the `closeParser()` method. The `com.ibm.di.parser.ParserImpl` abstract class provides default implementation for closing the Parser input and output streams.

public void setOutputStream(OutputStream os)
: Set the output stream of the Parser here. The output stream is passed as an argument. This method has an overloaded version that takes a "Writer" object as an argument. Like "setInputStream(...)" the abstract class `com.ibm.di.parser.ParserImpl` provides implementation for both methods that sets the output stream of the Parser. If you subclass `com.ibm.di.parser.ParserImpl` you can override some of these methods or leave the default implementation of the super class. However, if you implement the interface you will have to provide implementation for all of them.

  Note that when you open the output stream, it is your responsibility to close it. This is usually done in the `closeParser()` method. The `com.ibm.di.parser.ParserImpl` abstract class provides a default implementation for closing the Parser input and output streams.

public void initParser()
: Put any initialization code here. This method is usually called by the hosting component (for example, a Connector) to initialize the Parser.

  You can allocate resources you may need in future, as well as setting any parameters or additional chained parsers. This method may not be required for all implemented parsers.

  Here is an example of how you can access parameters. This set of code is part of the included example "ExampleParser.java".

```
str = getParam("attributeName");
    if (str != null && str.trim().length() != 0) {
     attrName = str;
    }
```

  Note that this method is called after setting of input and output streams is done.

public Entry readEntry()
: This method is similar to the "getNextEntry()" method used to implement a Connector. It returns

the next entry from the current input stream. In case that the input stream is depleted a null value is expected to be returned. This is the place where the actual parsing takes place.

Make sure you have initialized the input stream properly. In order to set the input stream you can use the `setInputStream(...)` method. You can use the `getReader()` method to get the reader object.

Generally input streams are initialized by the hosting component (for example, a Connector).

public void writeEntry(Entry entry)
Use this method to write an entry using the current output stream. The entry that will be written is passed as an argument. This is the place where you have to parse the data of the entry and write it in the proper format to the output stream.

In order to get the writer you can use the `getWriter()` method which returns a "java.io.BufferedWriter".

Generally the output stream is initialized by the hosting component (for example, a Connector).

public void flush()
This method is usually called by some hosting components to flush any in-memory data to the current output stream. So, to make sure that all the data is written and there is nothing left in the memory call this method.

public void closeParser()
This method is called by the hosting component (for example, a Connector) to close and release the Parser resources. Use this method to close and release any resources that may no longer be used and when the Parser will not be invoked anymore. In most cases this would be the input and output stream but if any additional resources were used, they should be released as well.

The `com.ibm.di.parser.ParserImpl` abstract class provides an implementation for this method but if you implement the interface you will have to write it by yourself.

Building the Parser source code

When building the source code of your Parser, include in your CLASSPATH the jar files from the "jars" folder of the IBM Tivoli Directory Integrator installation. As a minimum, you would need to include "miserver.jar" and "miconfig.jar". Keep in mind that the source code must be compiled for Java 5 or older.

Note: When integrating your Java code with Tivoli Directory Integrator, pay attention to the collection of pre-existing components that comprise Tivoli Directory Integrator, notably in the jars directory. If your code relies upon one of your own library components that overlap or clash with one or more that are part of the Tivoli Directory Integrator installation there will most likely be loader problems during execution. In other words, you should be careful about possible conflicts with third-party libraries that are shipped with Tivoli Directory Integrator. This means that you should avoid creating a Parser that uses one version of a library when Tivoli Directory Integrator uses another version of the same library.

Implementing the Parser GUI configuration form

The Parser GUI is implemented in the same way as for a Connector. Use a "tdi.xml" file to describe the Parser configuration form by using the same syntax as used for Connectors.

Packaging and deploying the Parser

Packaging and deploying a Parser is just like packaging and deploying a Connector:

You need a jar file that contains:

1. The class file(s) of the Parser

2. "tdi.xml" file placed in the root of the jar file. Note that this file is used to register the Parser. Without it the code will not be loaded, and you will not be able to use the Parser in the Tivoli Directory Integrator Configuration Editor (however you will be able to call it from scripts).

After you have created the jar file of the new Parser, you only need to drop that jar file in the "jars" folder in the Tivoli Directory Integrator installation. You can create your own folder and put the jar there but the general place where parsers are stored is the "jars/parsers" folder. The next time the Tivoli Directory Integrator is started it will automatically load the new Parser and it will be ready for use.

Creating additional Loggers

Traditionally, logging in IBM Tivoli Directory Integrator is accomplished by means of Server- or task (AssemblyLine)-based Appenders, which rely upon the Apache Log4J framework to do the actual log output. In Tivoli Directory Integrator 7.1, the hardwired link between Tivoli Directory Integrator and Log4J is severed, and replaced with a configurable logging class which by default invokes Log4J. While Log4J provides a variety of output channels and formats, there are other logging utilities with overlapping and additional output channels that you as a Tivoli Directory Integrator user may need. Many of these are open source libraries that are not bundled with Tivoli Directory Integrator for legal reasons. To enable inclusion of these 3rd party logging utilities, the Tivoli Directory Integrator logging component is modeled to act as a proxy between Tivoli Directory Integrator and the actual logging implementations, called `LogInterface` implementations. Tivoli Directory Integrator comes with implementations for Log4J, JLOG and java.util.log, as follows:

Table 90.

Logging Utility	**Handlers/Appenders**
Apache Log4J	Category based configuration *) ConsoleAppender CustomAppender DailyRollingFileAppender FileAppender NTEventLog FileRollerAppender SystemLogAppender SyslogAppender
Standard Java Logging (java.util.log)	FileHandler
JLOG	Category based configuration *) FileHandler

*) Category based configuration means that the configuration of the logger is defined in an external file specific to the logging utility such as "`log4j.properties`" for Log4J.

The TDI configuration file structure accommodates a top-level folder which holds `com.ibm.di.config.interfaces.LogConfigItem` objects. This folder is populated by the TDI class loader (IDILoader) scanning all jar and zip files for "tdi.xml" files. The "tdi.xml" files define new loggers that become available to the TDI user/CE by including appropriate sections. Each logger defined this way will also include a form definition that the TDI CE can present to the user for configuration of its custom logger parameters.

The `com.ibm.di.log.Log` class is designed to use one or more of these new log components. Each log component implements `com.ibm.di.log.LogInterface`. This class is responsible for mapping LogInterface methods to the corresponding methods in its logging utility framework. The log component is given the LogConfigItem object to properly configure its back end logger when it is instantiated.

Understanding the logging interface

The TDI logging interface consists of the following files and objects:

Table 91.

Object	Description
com.ibm.di.config.interfaces.LogConfigItem	This is the configuration object for a defined LogInterface implementation.
com.ibm.di.log.LogInterface	This is the interface implemented by loggers that provide access to 3rd party logging utilities.
com.ibm.di.server.Log	This is the utility class used by TDI components to create the Log object.
<workdir>/logging.categories	Optional file to map categories to LogInterface class names. This file is not present by default.
com.ibm.di.log.TDILog4j	The LogInterface implementation for Log4J.
<installdir>/etc/log4j.properties	The log4j configuration file for TDI main component categories (server,ce and config drivers).
<installdir>/etc/global.properties	A property is defined to globally enable/disable log activities. When false, all log calls made through the TDI Log class will be discarded. The property name is "`com.ibm.di.logging.enabled`".

TDI components obtain loggers by creating an instance of the `com.ibm.di.server.Log` class using a category to determine the actual logging utility and output to use. This is done to preserve backward compatibility.

The Log class will also consult the `logging.categories` file to see if there is a mapping between the category and a LogInterface class. By default there are no specific mappings in this file (it is not present by default) and the Log class falls back on the TDILog4J implementation.

Log Interface Configuration

The logging interface will look for a file called `logging.categories` in the working directory to override use of the default TDILog4J `LogInterface` implementation. Each line in this file contains a category name with a value giving the java class name of the LogInterface implementation.

For example:

```
*:com.ibm.di.log.TDILog4j
AssemblyLine.AssemblyLines/myAL:com.ibm.di.log.TDILogJUL
```

In the above example all AssemblyLines will log using the TDILog4j framework, except for the AssemblyLine named myAL that will use TDILogJUL. The logging utilities currently available are:

- Log4j – `com.ibm.di.log.TDILog4J`
- Java Util Logging – `com.ibm.di.log.TDILogJUL`
- JLOG – `com.ibm.di.log.TDIJLog`

Logger External Configuration

Logging utilities are typically configured using an external configuration file. For example, Log4J uses a property style file where names (categories) are mapped to specific output types/formats (for example, file output, XML format). Users then ask for a logger instance providing a category name, which in turn is resolved by the logging utility consulting its configuration file.

Each LogInterface implementation may require a properties file to provide correct configuration of its loggers. Specifically, if the default TDILog4j implementation is replaced by another logging utility, the new logging utility should provide loggers for those categories that are defined in the released version of the `etc/log4j.properties` file.

Logger Internal Configuration

The main components in Tivoli Directory Integrator use category names to obtain loggers (server, CE, config drivers and so forth) and rely on the external configuration to provide details about each logger. However, users can specify additional loggers using the configuration editor. The user may add loggers at the AssemblyLine level and/or at the server level. When the user adds a logger this way, the details about the logger is stored in the TDI configuration file. The instantiation of the logger is now handled by TDI and not by the logging facility and its external configuration file. Traditionally, TDI had a list of predefined Log4J loggers the user could choose from (see Table 91 on page 605). In IBM Tivoli Directory Integrator 7.1, the hard coded list of loggers has been externalized into the system namespace (Loggers folder).

Logger configurations are stored in the top-level folder **Loggers** in the configuration file. System wide available loggers are defined in the system namespace, which is built from `tdi.xml` files when TDI starts.

The `tdi.xml` files found in jar/zip files in the *installdir*`/jars` directory (and other custom loader directories) are added to the system namespace by the TDI loader (IDILoader). A logging component is defined by two separate sections in this file.

The loggers section defines the `LogInterface` implementation and parameters that designate a specific logger for a specific logging utility (for example, log4j, file logger). This section also contains a second parameter that points to a form definition used to present the configurable parameters for the user. Additional parameters may appear in this section specific to each logger.

```
<Logger name="ibmdi.JavaUtilLoggingFile">
 <parameter name="categoryBased">false</parameter>
 <parameter name="com.ibm.di.formName">ibmdi.JavaUtilLoggingFile</parameter>
 <parameter name="com.ibm.di.log.interface">com.ibm.di.log.TDILogJUL</parameter>
 <parameter name="handler">FileHandler</parameter>
</Logger>
```

The form section defines the configurable parameters for the logger.

```
<Form name="ibmdi.JavaUtilLoggingFile">
 <FormItemNames>
   <ListItem>fileName</ListItem>
  <ListItem>formatter</ListItem>
  ... other parameters...
 </FormItemNames>
 <FormItem name="fileName">
   <parameter name="description">The pattern for the log file name</parameter>
   <parameter name="label">File Name</parameter>
    <parameter name="Required">true</parameter>
   <parameter name="script">selectFile</parameter>
   <parameter name="scriptLabel">Select...</parameter>
    <parameter name="scripthelp">Choose the file name to use</parameter>
  </FormItem>
  <FormItem name="formatter">
  <Values>
   <ListItem>Simple</ListItem>
    <ListItem>XML</ListItem>
  </Values>
   <parameter name="description">Choose a SimpleFormatter or a XMLFormatter</parameter>
   <parameter name="label">Formatter</parameter>
   <parameter name="syntax">droplist</parameter>
  </FormItem>
  .... Other FormItem definitions
</Form>
```

The overall syntax for the tdi.xml file would be something like the following skeleton example:

```
<?xml version="1.0" encoding="UTF-8"?>
<MetamergeConfig>
 <Folder name="Loggers">
  <Logger name="...">
  </Logger>
  </Folder>
  <Folder name="Forms">
  <Form name="...">
  </Form>
 </Folder>
</MetamergeConfig>
```

Logger API

Adding a new logging utility to TDI involves creating a LogInterface implementation and providing a tdi.xml file with proper sections (see section "Logger Internal Configuration" on page 606). The implementation must also provide a static method to bootstrap a new logger.

com.ibm.di.server.Log: The main logging class in TDI has two methods defined to govern the logging activity on a global basis. The initial setting of the activity is defined by a property named com.ibm.di.logging.enabled. Logging that bypasses this class will not be affected.

```
/**
 * Disables or enables TDI logging. All loggers are affected by this setting.
 */
 public static void setLoggingEnabled(boolean enabled);

/**
 * Returns whether TDI logging is active or disabled.
 */
 public static boolean isLoggingEnabled(boolean enabled);
```

When logging is turned off from the start (for example, property is set to false) there may still be a few lines logged by TDI during initialization of the loader and the main program. If you want to remove logging completely, you should modify the logging utility's configuration file (for example, log4j.properties, jlog.properties etc).

com.ibm.di.log.LogInterface:

```
package com.ibm.di.log;


/**
 * Defines an Interface to new Loggers.
 * Any Logger we use must adhere to this interface.
 * The Implementation must provide a public constructor with no arguments.
 * After construction either the setCategory() or the addAppender() method will be called.
 */
public interface LogInterface {

 public final static String TYPE = "type";
 public final static String NAME = "name";
 public final static String CONFIG_INSTANCE = "configInstance";
 public final static String TIME = "time";

 /**
  * Set the category for this Logger.
  * This method specifies a category, to allow a category based configuration.
  *
  * @param category The category to use.
  */
 public void setCategory(String category) throws Exception;

 /**
  * Add an Appender to the Logger using the given config. Appender is the
```

```
 * org.apache.log4j name, java.util.logging would call it a Handler. May
 * throw an Exception if the config does not make sense.<br/>
 * The params Map may contain these keys to help set up the Appender:
 *
 * - TYPE: "AssemblyLine", "EventHandler" or ""
 * - NAME: A String with the name of component
 * - CONFIG_INSTANCE: a RSInterface
 * - TIME: a String with the time in milliseconds
 *
 *
 * @param config
 *            The LogConfigItem.
 * @param params
 *            Extra information that may be useful/
 */
public void addAppender(LogConfigItem config, Map params) throws Exception;

/**
 * Log a message with level debug.
 * @param str The string to be logged
 */

public void debug( String str );

/**
 * Log a message with level info.
 * @param str The string to be logged
 */

public void info( String str );

/**
 * Log a message with level warning.
 * @param str The string to be logged
 */

public void warn( String str );

/**
 * Log a message with level error.
 * @param str The string to be logged
 */

public void error( String str );

/**
 * Log a message with level error, and an additional Throwable.
 * @param str The string to be logged
 * @param error The Throwable to be logged
 */

public void error( String str, Throwable error );

/**
 * Log a message with level fatal.
 * @param str The string to be logged
 */

public void fatal ( String str);

/**
 * Log a message with level fatal, and an additional Throwable.
 * @param str The string to be logged
 * @param error The Throwable to be logged
 */

public void fatal ( String str, Throwable error );
```

```
 /**
  * Log a message with the specified level.
  * @param level The level to use when logging.
  * @param str The string to be logged
  */

 public void log (String level, String str );

 /**
  * Check if a debug message would be logged.
  * @return true if a debug message might be logged
  */
 public boolean isDebugEnabled ();


 /**
  * Free up all resources this logger uses.
  * The logger will not be called anymore.
  */
 public void close();
}
```

com.ibm.di.server.Log: TDI components that require logging capabilities should obtain a logger through the `com.ibm.di.server.Log` class. This class is a proxy between the client and the actual logging implementation. When TDI components add logging to their code, it should decide whether to reuse the existing logging categories that are predefined, create a new category or maintain its own LogConfigItem configuration object. In either case, it should end up using a Log object to do the actual logging.

Constructor and configuration: You typically use one of the two constructors to configure the logger. After the logger has been created you can use the `setPrefix()` method to set a string which is prefixed to all outgoing messages. The prefix string is not translated.

The category name is used to configure the logger. This is defined in the properties file for the logging utility in use. The resourceFileName is the name of a resource (loaded by `com.ibm.di.server.ResourceHash`) that contains an NLS table used when translating messages.

```
/**
 * Create a log object using category as both the name of the resource
 * file and the logger category name (configuration).
 */
public Log(String category);

/**
 * Create a log object using separate values for category and resource name.
 */
public Log(String category, String resourceFileName);


/**
 * Sets a prefix to be prefixed to all messages
 *
 * @param  prefix
 */
public void setPrefix (String prefix);
```

Simple log methods: There is a set of logging methods for the various levels provided as a convenience.

```
public void log<level>(String msg)
```

where **<level>** is the logging level:

- fine
- debug
- info

- warn
- error
- fatal

These methods will log the message (including any prefix) as-is to the logger.

```
log.setPrefix("PRE");
log.loginfo("Hello");

>> PRE Hello
```

NLS log methods: When you need to translate log messages you should use the following methods:

- fine(String resid)
- debug(String resid)
- info(String resid)
- warn(String resid)
- error(String resid)
- fatal(String resid)

where **resid** is the resource identifier in the resource file associated with the Log object. If a translation for the resource id isn't found, the resource identifier is used as-is in the log output. Each of these methods comes in four variants to let you supply values for substitution markers in the translated string. When you use one of the variants with substitution values, the Log class will use `java.util.text.MessageFormat` on the string with the parameters you provide. Three methods let you provide one, two or an arbitrary number of substitution values.

```
public void debug(String res)
public void debug(String res, Object param)
public void debug(String res, Object param1, Object param2)
public void debug(String res, Object[] params)
```

If you want to log your own error message with a Throwable object, you can use the method `error(String resid, Throwable error)`.

If you want to generate an exception with a translated message, you can use the method `exception(String resid) throws Exception` which will throw a generic java.lang.Exception object with the translated string as its message.

If you want to translate a string to use somewhere else, you can use the `getString()` methods to obtain the translated string from the resource file associated with the log object.

Example: This very simple example shows how to log an NLS message based on a properties file to the standard TDI server log (miserver). It is assumed that "`XXX.properties`" is packaged with the code.

Contents of "XXX.properties":

```
my.resource.id= Hello World
```

```
import com.ibm.di.server.Log;

public class XXX() {
 public XXX() {
  this.log = new Log("miserver";, "XXX");
  this.log.info("my.resource.id");
 }
}
```

The above should result in a log message "Hello World" written to the TDI server log.

See also

Appendix C, “Server API,” on page 535,
“Log Connector” on page 193

Appendix F. Notices

This information was developed for products and services offered in the U.S.A. IBM might not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department MU5A46
11301 Burnet Road

Austin, TX 78758
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Third-Party Statements

ICU License - ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995-2006 International Business Machines Corporation and others

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM	Tivoli	AIX®	Lotus
Notes	pSeries®	DB2	WebSphere
S/390®	Domino	iNotes	Cloudscape

Java, JavaScript and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Microsoft, Windows, Windows NT and the Windows logo are registered trademarks of Microsoft Corporation.

Intel® is a trademark of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the U.S., other countries, or both.

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

Other company, product, and service names may be trademarks or service marks of others.

Program Number: 5724-K74

Printed in USA

SC27-2707-00