Tivoli® Directory Integrator

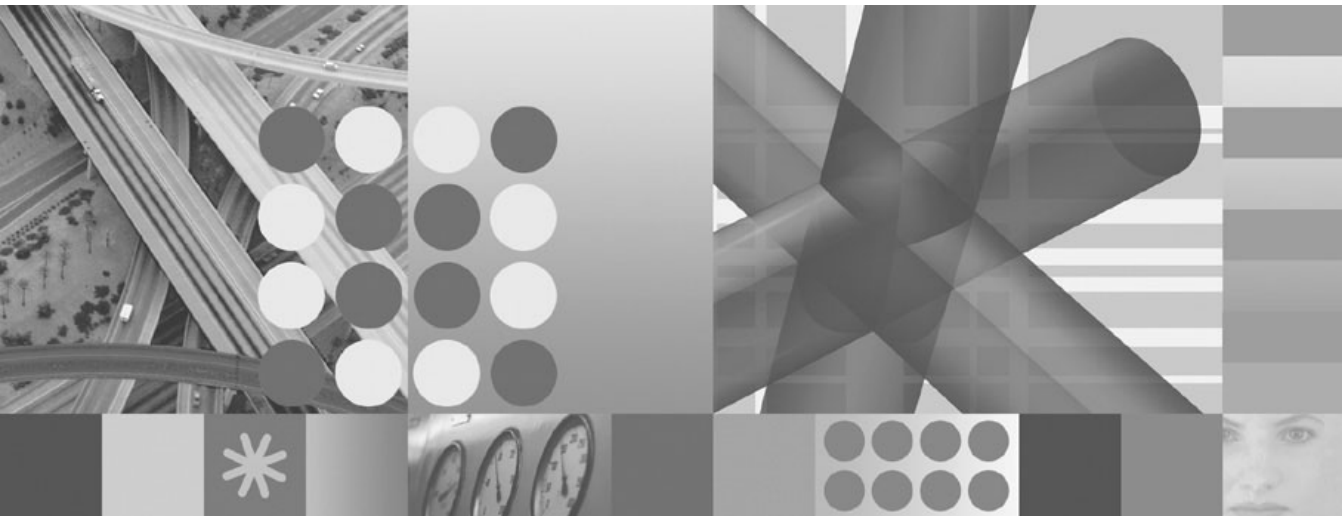
IBM

IBM Tivoli Directory Integrator 6.1: Reference Guide

SC32-2566-00

Tivoli® Directory Integrator

IBM



IBM Tivoli Directory Integrator 6.1: Reference Guide

SC32-2566-00

Note

Before using this information and the product it supports, read the general information under Appendix E, "Notices," on page 451.

First Edition (April 2006)

This edition applies to version 6.1 of the IBM Tivoli Directory Integrator and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2002, 2006. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

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.

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 R/3.

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 IBM Tivoli Directory Integrator library are:

IBM Tivoli Directory Integrator 6.1: Getting Started
A brief tutorial and introduction to IBM Tivoli Directory Integrator 6.1.

© Copyright IBM Corp. 2002, 2006

IBM Tivoli Directory Integrator 6.1: Administrator Guide
Includes complete information for installing the IBM Tivoli Directory Integrator. Includes information about migrating from a previous version of IBM Tivoli Directory Integrator. Includes information about configuring the logging functionality of IBM Tivoli Directory Integrator. Also includes information about the security model underlying the Remote Server API.

IBM Tivoli Directory Integrator 6.1: Users Guide
Contains information about using the IBM Tivoli Directory Integrator 6.1 tool. Contains instructions for designing solutions using the IBM Tivoli Directory Integrator tool (**ibmditk**) or running the ready-made solutions from the command line (**ibmdisrv**). Also provides information about interfaces, concepts and AssemblyLine/EventHandler creation and management. Includes examples to create interaction and hands-on learning of IBM Tivoli Directory Integrator 6.1.

IBM Tivoli Directory Integrator 6.1: Reference Guide
Contains detailed information about the individual components of IBM Tivoli Directory Integrator 6.1 AssemblyLine (Connectors, EventHandlers, Parsers, Plug-ins, and so forth).

IBM Tivoli Directory Integrator 6.1: Problem Determination Guide
Provides information about IBM Tivoli Directory Integrator 6.1 tools, resources, and techniques that can aid in the identification and resolution of problems.

IBM Tivoli Directory Integrator 6.1: Messages Guide
Provides a list of all informational, warning and error messages associated with the IBM Tivoli Directory Integrator 6.1.

IBM Tivoli Directory Integrator 6.1: Password Synchronization Plug-ins Guide
Includes complete information for installing and configuring each of the four IBM Password Synchronization Plug-ins: Windows Password Synchronizer, Sun ONE 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 MQe Password Store.

IBM Tivoli Directory Integrator 6.1: Release Notes
Describes new features and late-breaking information about IBM Tivoli Directory Integrator 6.1 that did not get included in the documentation. IBM Tivoli Directory Integrator 6.1.

Related publications

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

- IBM Tivoli Directory Integrator 6.1 uses the JNDI client from Sun Microsystems. For information about the JNDI client, refer to the *Java Naming and Directory Interface™ 1.2.1 Specification* on the Sun Microsystems Web site at http://java.sun.com/products/jndi/1.2/javadoc/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, in English only, from the **Glossary** link on the left side of the Tivoli Software Library Web page http://www.ibm.com/software/tivoli/library/

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 the **Fit to page** check box 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 software products successfully. With this product, 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.

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/

If you need additional help, contact software support by using the methods described in the *IBM Software Support Guide* 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

Contents

Preface iii
Who should read this book iii
Publications iii
IBM Tivoli Directory Integrator library . . iii
Related publications iv
Accessing publications online v
Accessibility v
Contacting IBM Software support v

Chapter 1. Introduction 1

Chapter 2. Connectors 3
Connector availability and reference 3
Connector Interfaces 3
Script-based Connectors 6
Configurations 7
Active Directory Changelog (v.2) Connector . . 7
Tracking changes in Active Directory . . . 7
Behavior 9
Using the Active Directory Changelog V2 Connector. 10
Configuration 11
Migration from Active Directory Changelog EventHandler to Active Directory Changelog (v.2) Connector . . . 13
See also 13
AssemblyLine Connector. 14
Configuration 14
Using the Connector 15
Axis Easy Web Service Server Connector . . 15
Hosting a WSDL file 17
Configuration 17
Connector Operation 19
See also 20
Btree Object DB Connector 20
Configuration 21
Btree object 21
Command line Connector 22
Some words on quoting 22
Configuration 23
Examples 23
See also 23
Direct TCP /URL scripting 23
TCP. 23
URL 24
Domino/Lotus Notes Connectors 24
Session types. 24
Java Class loader issue 26
Supported versions of Lotus Notes and Lotus Domino 27
Native API call threading 27
The ncso.jar file 27
Domino Change Detection Connector . . 29
Domino Users Connector 39
Lotus Notes Connector 56
DSMLv2 SOAP Connector 60
Supported Connector Modes 60
Extended Operations 61
Configuration 62
DSMLv2 SOAP Server Connector 63
Extended Operations 63
Configuration 64
Exchange Changelog Connector 65
Behavior 66
Using the Exchange Changelog Connector 66
The Is-Deleted attribute in Exchange . . . 67
Accessing the USN synchronization values in the User Property Store 69
Accessing the runtime Connector's USN synchronization values 70
Configuration 70
Migration 72
See Also 72
File system Connector. 73
Configuration 73
See also 74
FTP Client Connector 74
Configuration 74
See also 75
HTTP Client Connector 75
Modes 75
Special attributes 76
Configuration 77
Examples 78
See also 78
Old HTTP Client Connector. 78
Modes 79
Special attributes 79
Configuration 80
Examples 81

© Copyright IBM Corp. 2002, 2006

See also 81
HTTP Server Connector 81
Connector structure and workflow . . . 82
Connector Client Authentication 82
Chunked Transfer Encoding. 83
Configuration 83
Connector Schema 85
See also 86
Old HTTP Server Connector 86
Configuration 87
See also 87
IBM Directory Server Changelog Connector 87
Configuration 88
See also 90
ITIM Agent Connector 90
Setting up SSL for the ITIM Agent Connector. 90
Configuration 90
JDBC Connector 92
Connector structure and workflow . . . 93
Customizing select, insert, update and delete statements 94
Configuration 96
Link Criteria 98
Other 98
Timestamp 99
Calling Stored Procedures 99
SQL Databases: column names with special characters 100
Using prepared statements. 100
On Multiple Entries 100
ODBC–specifying database paths directly 101
JMS Connector. 101
Introduction 101
JMS message flow 102
WebSphere MQ and JMS/non-JMS consumers of messages 103
JMS message types 103
Iterator mode 105
Lookup mode 105
Add Only mode 106
Call/Reply mode 106
JMS headers and properties 106
Configuration 108
Examples. 111
External System Configuration 111
JMX Connector. 112
Connector Schema 112
Configuration 113
JNDI Connector 114
Configuration 115
Setting the Modify operation 116
See also 118
LDAP Connector 118
Configuration 120
Virtual List View Control 123
Handling memory problems in the LDAP Connector 124
LDAP Connector methods (API) 124
See also 127
LDAP Server Connector. 127
Scripting 128
Returning the LDAP message returned values. 128
Error handling 128
Configuration 128
See also 129
Lotus Notes Connector 129
Mailbox Connector 129
Configuration 130
Predefined properties and attributes . . 131
See also 132
Memory Queue Connector. 132
Memory queue components 133
High level workflow. 134
Configuration 134
Accessing the Memory Queue programmatically 135
Memory Stream Connector. 136
Configuration 136
MQe Password Store Connector 137
MQe Password Store Connector Entry structure 138
Configuration 138
See Also 139
Netscape/iPlanet Changelog Connector . . 139
Configuration 140
See also 142
Server Notifications Connector 142
Encryption and Cryptography 142
Authentication. 143
Configuration 143
System Properties Connector 145
Configuration 145
System Queue Connector 146
Introduction 146
Configuration 147
Security and Authentication 148
MQe Initialization. 148
Windows Users and Groups Connector . . 149

Preconditions 149
Character sets 153
Examples 153
Windows Users and Groups Connector functional specifications and software requirements 153
System Store Connector 154
Configuration 155
Using the Connector 157
See also 158
RDBMS Changelog Connector 158
Configuration 159
Change table format 160
Creating change tables in DB2 161
Creating change tables in Oracle 162
Creating Change table and triggers in MS SQL 163
Creating change table and triggers in Informix 165
Creating change table and trigger for SYBASE 169
runtime-provided Connector 171
Configuration 172
See also 172
Script Connector 172
Predefined script objects 173
Functions 173
Configuration 175
Examples 175
See also 175
SNMP Connector 175
Configuration 175
Examples 176
SNMP Server Connector 176
Connector Schema 177
Configuration 178
TCP Connector. 178
Iterator Mode 178
AddOnly Mode 179
Configuration 179
See also 180
TCP Server Connector 180
Configuration 180
Connector Schema 181
See Also 182
Timer Connector 182
Configuration 182
URL Connector 182
Configuration 183
Supported URL protocol 183
See also 183
Web Service Receiver Server Connector . . 183
Hosting a WSDL file 184
Configuration 184
Connector Operation. 186
See also 187
z/OS Changelog Connector 187
Configuration 187
See also 189

Chapter 3. EventHandlers 191
Migration from ChangeLog EventHandlers to ChangeLog Connectors 191
EventHandler types 192
When are they started? 192
What do they do?. 192
Data flow 193
Passing input/output file names to an AssemblyLine 193
EventHandler availability 193
Migration of Changelog EventHandlers . . 194
Active Directory Changelog EventHandler 194
Behavior 195
Access to the USN synchronization values in the User Property Store 195
Access to the runtime EventHandler's USN synchronization values 196
Configuration 196
See also 198
Connector EventHandler 198
Configuration 198
Objects/properties/attributes 198
See also 198
DSMLv2 EventHandler 199
Transportation (binding) 199
EventHandler Workflow 199
Operations 200
Configuration 200
Exchange Changelog EventHandler 202
Behavior 203
Access to the USN synchronization values in the User Property Store 203
Access to the runtime EventHandler's USN synchronization values 204
Configuration 204
See also 206
HTTP EventHandler 206
Example 206
Configuration 207
See also 208

IBM Directory Server EventHandler. . . . 208
Configuration 209
See also 210
LDAP EventHandler 211
Object Added (_objAdded). 211
Object Rename (_objRenamed) 212
Object Modified (_objModified) 212
Object Removed (_objRemoved) 212
Error Encountered (_handleError) . . . 212
Configuration 213
See also 214
LDAP Server EventHandler 214
Scripting 214
Returning the LDAP message returned values. 215
Error handling 215
Configuration 215
Mailbox EventHandler 216
Configuration 216
Objects/properties/attributes 216
Examples 217
See also 217
SNMP EventHandler. 217
Scripting the desired action 218
Error handling 218
Returning the SNMP packet returned values. 219
Configuration 219
TCP Port EventHandler 220
Configuration 220
Objects/properties/attributes 220
Examples 221
See also 221
Generic thread (primitive EventHandler) . . 221
Configuration 221
See also 221
Timer EventHandler (primitive EventHandler) 221
Configuration 221
Examples 222
zOS LDAP Changelog EventHandler . . . 223
Configuration 223
Polling logic 224
See also 225

Chapter 4. Parsers 227
Base Parsers 227
Character Encoding conversion 227
Availability 228
CSV Parser 228
Configuration 228
DSML Parser 229
Configuration 229
Examples 229
See also 230
DSMLv2 Parser 230
Mode 231
Operations 231
Binary and non-String Attributes. . . . 237
Optional Attributes 237
Setting result code and result description 238
Multiple Attribute modifications 238
Configuration 239
Examples 240
Fixed Parser 242
Configuration 242
HTTP Parser 242
Configuration 242
Attributes or properties 243
Character sets/Encoding 244
See also 245
LDIF Parser. 245
Configuration 245
See also 246
Line Reader Parser 246
Configuration 246
Script Parser 247
Objects 247
Functions (methods) 248
Configuration 249
Example 249
See also 249
Simple Parser 250
Configuration 250
SOAP Parser 250
Example Entry 250
Example SOAP document 250
Configuration 251
Parser-specific calls 251
Examples 252
XML Parser 252
Configuration 253
Character Encoding in the XML Parser 254
Examples 254
Additional Examples. 256
See also 256
XML SAX Parser 256
Configuration 258
See also 258
XSL based XML parser 258

Introduction 258
Configuration 259
Using the Parser 259
See also 261
User-defined parsers 262

Chapter 5. Function Components 263
XMLToSDO FC 264
Example 264
Configuration 265
Migration 265
SDOToXML FC 266
Configuration 268
Using the FC 268
Migration 268
AssemblyLine FC 269
Configuration 269
Using the FC 270
Java Class Function Component 271
Schema 271
Configuration 272
Parser FC 272
Configuration 272
Using the FC 273
Scripted FC 273
Configuration 273
Using the FC 273
See also 274
CBE Generator Function Component . . . 274
Common Base Event (CBE) 274
The Common Event Infrastructure (CEI) 275
CBE FC Configuration 275
Input and Output Map Attributes . . . 275
Generating a CBE Log XML 277
See also 278
SendEMail Function Component 278
Configuration 278
Memory Queue FC 279
Configuration 279
Using the FC 280
See also 281
Axis Java To Soap FC 281
Configuration 281
Using the FC 282
WrapSoap FC 284
Configuration 284
Using the FC 285
InvokeSoap WS FC 286
Introduction 286
Authentication 286
Configuration 287
Using the FC 288
See also 290
Axis Soap To Java FC 290
Configuration 290
Using the FC 291
Axis EasyInvoke Soap WS FC. 291
Authentication 292
Configuration 292
Using the FC 293
See also 294
Complex Types Generator FC 294
Configuration 295
Function Component Input and Output 295
Troubleshooting 296
Remote Command Line FC 296
Configuration 296
Function Component Input 298
Function Component Output 298
Using the FC 298
See also 300
z/OS TSO/E Command Line FC. 300
Configuration 300
Using the FC 301
Setting up the native part of the FC. . . 303
See also 304

Chapter 6. SAP R/3 Component Suite . . 305
Who should read this chapter. 305
Component Suite Installation 305
Software Requirements 305
Verifying the Component Suite for SAP R/3 306
Checking the Version Numbers 307
Uninstallation 308
Function Component For SAP R/3 308
Function Component Introduction . . . 308
Configuration 309
Using the Function Component 312
User Registry Connector for SAP R/3 . . . 313
Introduction 313
Configuration 315
Using the User Registry Connector for SAP R/3 319
Human Resources/Business Object Repository Connector for SAP R/3 323
Introduction 323
Configuration 327
Using the Human Resources Connector for SAP R/3 330

Troubleshooting the SAP R/3 Component Suite 336
Supplemental information for the SAP R/3 Component Suite 338
Example User Registry Connector XML Instance Document 338
XSchema for User Registry Connector XML 341

Chapter 7. Script languages 353
JavaScript 353
Java and JavaScript 353

Chapter 8. Objects 355
The AssemblyLine Connector object. 355
The attribute object 355
Examples 356
See also 356
The Connector Interface object 356
Methods 356
The Entry object 357
Global Entry instances available in scripting 357
See also 358
The FTP object 358
Example 358
Main object 359
The Search (criteria) object 359
Operands 359
Example 359
The shellCommand object 359
The status object 360
The system object 360
The task object 360

Appendix A. Password Synchronization Plug-ins 361

Appendix B. AssemblyLine and Connector mode flowcharts 363
AssemblyLine flow 364
Connector initialization 366
Close flow 367
AddOnly mode 368
Call/Reply mode 369
Delete mode 370
Delta Mode 372
Iterator mode 376
Lookup mode 377
Server Mode 378
Update mode 380
End-of-flow for all modes 383
Connector Reconnect. 384
Function Components 385

Appendix C. Server API 387
Overview 387
Sample use case 388
Local and Remote Server API interfaces . . 389
Server API structure 389
Security 390
Configuring the Server API 391
Configuring the Server API properties 391
Setting up the User Registry 391
Remote client configuration 391
Using the Server API 393
Creating a local Session 393
Creating a remote Session 393
Working with Config Instances 394
Working with AssemblyLines 395
Working with EventHandlers 399
Editing configurations 399
Working with the System Queue 403
Working with the Tombstone Manager 404
Working with TDI Properties 408
Registering for Server API event notifications. 409
Getting access to log files 411
Server Info 413
Using the Security Registry 414
Custom Method Invocation 414
The JMX layer 416
Local access to the JMX layer 417
Remote access to the JMX layer 417
MBeans and Server API objects 418
JMX notifications 418
JMX Example - TDI 6.1 and MC4J configuration 419
Backward compatibility 423
Scenarios overview 423
Server API changes in TDI 6.1 426
Known issues 432

Appendix D. Implementing your own Components 433
Support materials for Component development 433
Developing a Connector 433
Implementing the Connector's Java source code 433

Building the Connector's source code . . 438
Implementing the Connector's GUI configuration form 438
Packaging and deploying the Connector 446
Developing a Function Component 447
Implementing Function Component Java source code 447
Building the Function Component source code 448
Implementing the Function Component GUI configuration form 448
Packaging and deploying the Function Component 448
See also 449

Appendix E. Notices 451
Trademarks 453

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.

© Copyright IBM Corp. 2002, 2006

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).

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.

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 Changelog (v.2) Connector" on page 7
I

"AssemblyLine Connector" on page 14
I

"Axis Easy Web Service Server Connector" on page 15
S

"Btree Object DB Connector" on page 20
A D I L U

"Command line Connector" on page 22
A I

"Direct TCP /URL scripting" on page 23
?????

"Domino Change Detection Connector" on page 29
I

"Domino Users Connector" on page 39
A D I L U

"DSMLv2 SOAP Connector" on page 60
A D I L U C Δ

© Copyright IBM Corp. 2002, 2006

“DSMLv2 SOAP Server Connector” on page 63
S

“Exchange Changelog Connector” on page 65
I

“File system Connector” on page 73
A I

“FTP Client Connector” on page 74
A I

“Old HTTP Client Connector” on page 78
A D I L U C

“HTTP Client Connector” on page 75
A I L C

“Old HTTP Server Connector” on page 86
A D I L U C

“HTTP Server Connector” on page 81
A I S

“Human Resources/Business Object Repository Connector for SAP R/3” on page 323
A D I L U

“IBM Directory Server Changelog Connector” on page 87
I

IBM MQ Connector
A I L C

“ITIM Agent Connector” on page 90
A D I L D

“JDBC Connector” on page 92
A D I L U Δ

“JMS Connector” on page 101
A I L C

“JMX Connector” on page 112
I

“JNDI Connector” on page 114
A D I L U

“LDAP Connector” on page 118
A D I L U Δ

“LDAP Server Connector” on page 127
S

“Lotus Notes Connector” on page 56
A D I L U

“Mailbox Connector” on page 129
I L D

“Memory Queue Connector” on page 132
A I

“Memory Stream Connector” on page 136
A I

“MQe Password Store Connector” on page 137
I

“Netscape/iPlanet Changelog Connector” on page 139
I

“System Store Connector” on page 154
A D I L U

“RDBMS Changelog Connector” on page 158
I

“runtime-provided Connector” on page 171
A D I L U C

A runtime provided Connector is a Connector sent to the AssemblyLine as a parameter when the AssemblyLine is started. We cannot say in advance what modes that Connector supports since you wrote it and the available modes are not visible to TDI. Also see “runtime-provided Connector” on page 171.

“Script Connector” on page 172
A D I L U C

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

“Server Notifications Connector” on page 142
A I

“SNMP Connector” on page 175
A I L

“SNMP Server Connector” on page 176
S

“System Properties Connector” on page 145
A I U L D

“System Queue Connector” on page 146
A I

“TCP Connector” on page 178
A I

"TCP Server Connector" on page 180
I S

"Timer Connector" on page 182
I

"URL Connector" on page 182
A I

"User Registry Connector for SAP R/3" on page 313
A D I L U

"Web Service Receiver Server Connector" on page 183
S

"Windows Users and Groups Connector" on page 149
A D I L U

"z/OS Changelog Connector" on page 187
I

Legend for the Supported Mode columns

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

Script-based Connectors

A source of problems can appear 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.

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
? ? ? ? ?

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 6.1: Users Guide*.

Configurations

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

Active Directory Changelog (v.2) Connector

The Active Directory Changelog (v.2) Connector (hereafter referred to as ADCLV2) 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 ADCLV2 Connector operates in Iterator mode.

Notes:

1. This Connector is a replacement for the Active Directory Changelog Connector; usage of the latter is deprecated.
2. This version of the Connector is able to process huge AD Servers (millions of entries) regardless of the administrative time limit for executing a query on AD (the MaxQueryDuration setting). In comparison the old version of the Connector could fail with TimeLimitExceeded error when run against big AD Servers.
3. It uses a simpler algorithm for retrieving changes and uses only one USN number to represent the synchronization state. In comparison the old Connector uses 4 USN numbers and a fairly complex algorithm.
4. It does not distinguish between ″add″ and ″modify″ operations - both are reported as ″modify″; delete operations are reported as ″delete″. Not being able to distinguish between ″add″ and ″modify″ is not a serious restriction because the TDI Update Connector mode natively handles ″add″ and ″modify″ operations.
5. It might report ″delete″ operations for entries that have not been added to the repository being synchronized with AD (this will happen when an entry is added and deleted in AD while the Connector has been offline). It is something to be aware of, but it is not a serious restriction because TDI 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.
6. The parameter **Page Size** specifies the size of the pages AD will return entries on (default value is 500).

Tracking changes in Active Directory

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

The ADCLV2 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 ADCLV2 Connector reads from the IBM Tivoli Directory Integrator's User Property Store this USN value stored from the most recent ADCLV2 Connector session.

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 ADCLV2 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.

Behavior

The ADCLV2 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 Connector might report ″delete″ operations for entries that have not been added to the repository being synchronized with Active Directory - this will happen when an entry is

added and deleted in Active Directory while the Connector has been offline. This is not a serious restriction because IBM Tivoli Directory Integrator'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.

The ADCLV2 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.

The ADCLV2 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 ADCLV2 Connector acts as if the object or the attribute does not exist in Active Directory.

Using the Active Directory Changelog V2 Connector

Each delivered entry by the Connector contains the **changeType** attribute whose value is either ″update″ (for newly created and 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 ADCLV2 Connector can be interrupted 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.

This Connector supports the IBM Tivoli Directory Integrator 6.1 Checkpoint/Restart functionality. When a restart is requested and restart data is passed, the Connector retrieves the USN number from the restart data and starts synchronization from this USN number.

Configuration

The Connector needs the following parameters:

LDAP URL
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
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
The credentials (password).

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

Authentication Method
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
This parameter 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
This 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
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 size of the pages AD will return entries on (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.

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
This 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
This 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
This updates the System Store with the change log number when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual
This 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 ADCLV2 Connector's *saveStateKey()* method, somewhere in your AssemblyLine.

Use Change Notifications

This 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.

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.

Migration from Active Directory Changelog EventHandler to Active Directory Changelog (v.2) Connector

You need to do the following to reproduce an old EventHandler's configuration into an ADCLv2 Connector's implementation:

1. Create a new AssemblyLine and insert the Active Directory Changelog(v.2) Connector in it.
2. Set the ldapUrl, ldapUsername, ldapPassword, ldapAuthenticationMethod, ldapUseSSL, ldapSearchBase and debug Connector parameters to the values of the corresponding EventHandler parameters.
3. Set the iteratorStateKey Connector parameter to persistentParameterName EventHandler parameter. 4.
4. Set the useNotifications Connector parameter to "true".
5. When implementing the AssemblyLine flow consider that the Connector reports newly added entries as modify.

See also

"LDAP Connector" on page 118,
"Exchange Changelog Connector" on page 65
"Netscape/iPlanet Changelog Connector" on page 139,
"IBM Directory Server Changelog Connector" on page 87
"z/OS Changelog Connector" on page 187.

AssemblyLine Connector

AssemblyLines are often called as compound functions from other AssemblyLines and EventHandlers. 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.

The AssemblyLine Connector supports Iterator mode only, except when calling another AssemblyLine which supports AssemblyLine Operations. See "AssemblyLine Operations" in UsersGuide for more information.

The server-server capability addresses security concerns when managers want TDI 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.

Note: The return signature of previous versions of TDI is different, hence it is not possible to call AssemblyLines on a remote server from this version (TDI 6.1) to previous versions, like TDI 6.0.

Configuration

The Connector needs the following parameters:

AssemblyLine
The name of the AssemblyLine to be executed from this Connector. Choose from the drop-down list or enter the name.

Remote Server
The TDI server on which to run the AssemblyLine. Use blank for local instance or *host[:port]*.

Config Instance
The config instance ID or URL 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 6.1: 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 TDI Server public certificate.

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

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.

Attribute Mapping

The AssemblyLine Connector's input attribute map provides the returned attribute(s) from the target AssemblyLine; those returned attributes are set up in the Call/Return section of the target AssemblyLine.

You can retrieve those attributes by means of the Query Schema button in the Input Map section; also, 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.

Axis Easy Web Service Server Connector

The Axis Easy Web Service Server Connector is part of the TDI Web Services suite. It is a simplified version of the "Web Service Receiver Server Connector" on page 183 in that it internally instantiates, configures and uses the AxisSoapToJava and AxisJavaToSoap FCs.

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 183. When using this Connector you forgo the possibility of hooking custom processing before parsing the SOAP request and after serializing the SOAP response, i.e. 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 TDI 6.1 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 TDI 6.1 configurations if there is an operation named "Default", the corresponding operation in the WSDL file is named as the name of the AssemblyLine.
- In TDI 6.1 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 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 gives 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, i.e. 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 TDI developer/deployer to make sure that a TDI solution handles this correctly. It is recommended that 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 TDI 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"* Attrribute, and thus to the web service client in case a request for the WSDL was received.

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.

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.

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: a comma, a semicolon, a space, a carriage return or a new line.

Tag Op-Entry

When this parameter is checked (i.e., "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 TDI 6.1 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, i.e. 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 TDI installation folder.

Require Client Authentication

This parameter specifies whether this Connector will require clients to authenticate with client SSL certificates. If the value of this parameter is **true** (i.e., 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

The Axis Easy Web Service Server Connector stores the following information from the HTTP/SOAP request into Connector Attributes, available to be mapped in (that is, if the request is a SOAP request):

- 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.
- The SOAP request message itself in text XML form – stored into the *″soapRequest″* Attribute.
- Whether the WSDL file was requested – in the *″wsdlRequested″* Attribute. In this case, the WSDL file is stored in the *″soapResponse″* Attribute, and none of the other Attributes are set.
- The parsed representation of the SOAP request in the *″requestObjArray″* Attribute.

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 TDI.

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 183.

Btree Object DB Connector

The Btree Connector is a simple database capable of storing Java objects. Each object is uniquely identified by a value called the key. The Connector uses an underlying Btree implementation to store AssemblyLine **Entry** objects. This enables the user to store the **conn** and **work** entries using a unique key. This Connector is also used by the AssemblyLine's Delta feature, although in version 5.2 and later of the product the usage of the System Store is recommended for this purpose.

If you want to use the Btree implementation directly to store a Java object other than AssemblyLine entries you must first get the Btree object and then use its methods directly.

Notes:

1. The "Btree Object DB Connector" on page 20 is **deprecated** with this release of IBM Tivoli Directory Integrator, and will be removed in a future version. Use the "System Store Connector" on page 154 instead, even for small data sets.
2. The "Btree Object DB Connector" on page 20 creates a new Btree database if one does not exist. However, if you iterate on a non-existing database, it is created and the Iterator returns no values.
3. The Btree Connector excels at small, quick jobs, but because the Btree Connector does not automatically balance its data structures, it breaks down when sorted lists are entered containing a few thousand entries. For randomly-ordered data sets the limits are somewhat higher. For larger data sets, consider using the "System Store Connector" on page 154, or the bundled CloudScape™ database using a "JDBC Connector" on page 92 (also see ″Using CloudScape database″ in *IBM Tivoli Directory Integrator 6.1: Users Guide*).

Configuration

The Connector needs the following parameters:

DB Filename
The file path where the Btree data is stored.

Key Attribute Name
The attribute name giving the unique value for the entry.

Selection Mode
Specify **All**, **Existing** or **Deleted**. In order to use the **Existing** and **Deleted** keywords, the Connector (database) must have been used by an AssemblyLine with the delta enabled. When Delta is enabled on an Iterator using the Btree method, the AssemblyLine stores a sequence property in the database and also adds a sequence number to each entry read from the source.

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

Btree object

The **getDatabase()** method returns the underlying Btree object. This object can be used to store other Java objects than AssemblyLine entries. The following snippet shows how you can insert, search and replace objects in the database:

```
var bt = system.getConnector("btreedb");
bt.initialze (null);

var db = bt.getDatabase();
db.insert ("my key", new java.lang.String("my value"));
var value = db.search ("my key");
value = value + " - modified";
db.replace ("my key", value);
```

Note: The BTree Connector lets you Lookup or Update on the keyAttribute only. Also, the BTree Connector does not support the Advanced Link Criteria (see ″Advanced link criteria″ in *IBM Tivoli Directory Integrator 6.1: Users Guide*).

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.

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, TDI uses the `sh` program (e.g. 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 TDI 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, an additional log message is generated.

Parser The Parser responsible for interpreting or generating entries.

Examples

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

See also

"Remote Command Line FC" on page 296,
"z/OS TSO/E Command Line FC" on page 300

Direct TCP /URL scripting

You might want to access URL objects or TCP ports directly, not using the Connectors. The following is example code 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. The decision on which client library to install, hinges on which Session Type is required.

Session types

Local Client Session

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

A Notes client must be installed locally. This session type requires `Notes.jar` file to be present in the <TDI_install_folder>/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 TDI is installed.

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

Provide the IOR String explicitly

The IOR is a text string required by the Domino Java API in order to establish an IIOP session to the Domino Server. The TDI 6.0 Domino Change Detection Connector uses a session creation method which obtains the IOR string from the Domino HTTP task. In TDI 6.1 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 create a session using the value of the **IOR** parameter. If SSL is to be used, the Connector uses the session creation method that accepts an array of strings as a parameter. The **HTTP Port** parameter is used only when the **IOR** parameter is empty.

If SSL is 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.

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

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

The port is appended to the host. The Domino HTTP task must run on this port without using SSL. The method will try to get the IOR string from the HTTP task and create an IIOP session.

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

Supported session types by Connector

Table 1. 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	No
Lotus Notes Connector	Yes	Yes	Yes

* The IIOP session connection type for the Domino Change Detection Connector is deprecated for this release, and may be removed in future versions of TDI.

Note: The Domino APIs for SSL are not JSSE compliant, and are instead Domino specific. This means that the TDI 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 TDI (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. Whether the TDI Truststore or Keystore are set or not in the `global.properties` (or `solution.properties`) is of no consequence to this Connector.

Java Class loader issue

Some of the classes in both the “Notes.jar” and “ncso.jar” libraries have exactly the same fully qualified Java class names. That is why if both jar files are in the Java class path, only one of

them is loaded by the Java class loader. Since it is undefined which one will be loaded, one normally removes one of the jar files from the class path and leaves only the needed one. In this way only one type of Notes/Domino application session can be used at a time from a TDI component, because switching the type of the application session supported requires stopping the TDI server, changing the jar file and then starting the TDI server again.

The Domino Users Connector uses only the "Notes.jar" library as it does not create IIOP sessions. That is why the "ncso.jar" library must not be in the Java class path. Thus other TDI components which need the "ncso.jar" library in order to establish an IIOP session **cannot run** while the Domino Users Connector is running.

Correspondingly when the Connector runs with IIOP sessions only "ncso.jar" must be presented in classpath and "Notes.jar" must be removed from it.

Supported versions of Lotus Notes and Lotus Domino

These Connectors are supported on Domino R6 and Domino R6.5

Native API call threading

When an AssemblyLine (containing Connectors) is executed by the TDI Server it runs in a single thread and it is only the AssemblyLine thread that accesses the AssemblyLine Connectors. The initializating 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 use is to call the NotesThread.sinitThread method when a local session is created.

That is why the Domino Change Detection Connector and Domino Users Connector 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 local session is created.

The ncso.jar file

In order to use IIOP sessions, the TDI 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 TDI product. You need to manually provide this file in order for the TDI 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 TDI_root\jars\3rdparty\IBM folder, so that the TDI Server will load it on initialization. Since the “ncso.jar” will not be provided as part of the IBM Tivoli Directory Integrator 6.1 installation, some existing TDI 6.0 functionalities will change as follows.

TDI Server

The “-v” command-line option: The TDI Server provides the “-v” command-line option which displays the versions of all TDI components. Since the “ncso.jar” file will not be provided as part of the TDI 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 TDI 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.

TDI Config Editor Aspects

Component version table: This is the table with the versions of all installed TDI components (available from menu ″Help″->″About IBM Tivoli Directory Integrator Components″). 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

Connector mode combo box: The Connector mode combo box will display all existing TDI 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.

Domino Change Detection Connector

The Domino Change Detection Connector enables IBM Tivoli Directory Integrator 6.1 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.

Note: Refer to 26 for an overview of which session types are possible with this Connector.

When run 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 as it works.

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, please see the *IBM Tivoli Directory Integrator 6.1: Password Synchronization Plug-ins Guide*.

The Connector stores locally, on the IBM Tivoli Directory Integrator 6.1 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. That 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.

The Connector supports two modes:

- *Assured once and only once delivery* – in this mode regardless of when the Connector is started and stopped and whether it was interrupted (by a system crash, etc.), it will report all changes (those that happened while the Connector was offline and those that are happening while it is online), once and only once, so that other repositories can be properly synchronized with Domino. In particular, when run in this mode, the Connector supports the IBM Tivoli Directory Integrator 6.1 Checkpoint/Restart functionality.
- *Normal assured delivery* – in this mode the Connector delivers all changed documents, but it will possibly report some changed objects twice in a case of interruption and restart. The checkpoint/restart functionality cannot be used when the Connector is run in this mode.

The *Normal assured delivery mode* is the more efficient in terms of speed and local storage. The *Assured once and only once delivery mode* is two to three times slower and requires local disk storage to keep the UnIDs of all documents processed on a single poll.

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 R6 and Domino R6.5, this parameter can be accessed from the Lotus Domino Administrator: open the database, then choose from the menu File | Replication | Settings..., 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 both on Domino R6 and Domino R6.5.

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 ″System Store 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.

The password prompt: A password prompt blocks the Connector (and thus the AssemblyLine) until the password of the local Notes ID file is typed in.

Note: If the ID file has no password, then no password prompt is displayed. Thus it is possible to run an AssemblyLine that contains the Domino Change Detection Connector from within the Config Editor (without the password prompt blocking it).

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, i.e. 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 is run in Assured once and only once delivery mode. When the Connector is run in *Normal assured delivery* mode the value of this Attribute is always ″false″.

When the Connector is run in *Assured once and only once delivery* mode 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 ***System Store 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 completes successfully 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 **System 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.

Running from the Administration and Monitor Console (AMC): When an AssemblyLine that contains the Domino Change Detection Connector is started from AMC it displays the Lotus Notes password prompt on the console of the IBM Tivoli Directory Integrator (TDI) Server. Since generally AMC and the TDI Server run on different machines, the AMC operator will not see this password prompt on the machine he or she works on. That is why the AMC operator might be tricked into thinking that the AssemblyLine is running properly, while at the same time it is waiting for the User ID password to be manually entered on the Server machine. Also, that is why when starting an AssemblyLine (that contains a Domino Change

Detection Connector) from AMC, the AMC operator must see to it that the User ID password be manually entered on the IBM Tivoli Directory Integrator Server machine for the AssemblyLine to execute properly.

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 huge Domino databases (.nsf files): The Connector could need a bigger amount of physical memory – for example when working on huge databases containing 1 000 000 documents or more, especially when performing a full synchronization. This is caused by the Connector keeping all the 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.

Checkpoint/Restart support: The Domino Change Detection Connector supports the TDI Checkpoint/Restart functionality only when run in *Assured once and only once delivery* mode. The *Assured once and only once delivery* mode keeps track of the complete state of the synchronization process and is therefore able to restart the synchronization process if interrupted, i.e. when run in this mode the Connector is capable of checkpoint/restart.

Required Setup of the IBM Tivoli Directory Integrator

See the section, 26 and the sections below about he 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 user 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 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 (i.e. ″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 (i.e. ″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 Local Client calls. This is a drop-down list; the default value is "IIOP". Note that the IIOP session connection type for this Connector is deprecated; future version of TDI will only support Local Client calls, at which time this parameter will be removed.

Domino Server IP Address
The IP address 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.

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

User Name
The name of the user used for the Java IIOP session authentication as specified by the first value of the User name field of the user’s Person document.

Internet Password
The password of the user used for the Java IIOP session authentication as specified in the ″Internet password″ field of the user’s Person document.

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

Delivery Mode
Specifies whether the Connector will run in ″Assured once and only once delivery″ or ″Normal assured delivery″ mode.

System Store 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 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 **Delete** button 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. 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 run once.)

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 **System Store 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 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 ″**System Store Key**″ is not found in the System Store and the ″**Start At**″ parameter is set to ″**Specific Date**″.

State Key Persistence

This 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

This updates the System Store when you read an entry from the Domino change log, before you continue with the rest of the AssemblyLine.

End of Cycle

This updates the System Store when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual

This 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, somewhere 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.

Use SSL

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

Detailed Log

Enable this for additional log messages.

Troubleshooting the Domino Change Detection Connector

1. **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 get IOR from Domino Server: ...* where *<domino_server_ip>* is the IP address of the Domino Server you are trying to access, i.e. 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, i.e. 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.

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 doing the following:

   Windows platforms

   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 (i.e. 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.

   This will have no effect if you are trying to run an AssemblyLine with a memory problem from the Config Editor (ibmditk), as the Config Editor starts a new instance of the JVM to run the AssemblyLine; with default parameters. In order to accommodate this situation, you need to do the following:

   a. Edit the global.properties or solution.properties file to alter the settings of com.ibm.di.javacmd to refer to a batch file. (for example, com.ibm.di.javacmd=c:\Program Files\IBM\TDI\V6.1\myjava.bat)
   b. Create a command file (the aforementioned c:\Program Files\IBM\TDI\V6.1\myjava.bat) containing the appropriate Java invocation command, like `"javaw" -Xms254m -Xmx1024M %*`

   Now the CE will use the modified JVM invocation, with increased heap size.

   Unix/Linux platforms

   Edit *ibmdisrv* in the IBM Tivoli Directory Integrator install directory to adjust the existing -Xms16m option to -Xms254m -Xmx1024m in the last line of the file (i.e. 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.

   This will have no effect if you are trying to run an AssemblyLine with a memory problem from the Config Editor (ibmditk), as the Config Editor starts a new instance of the JVM to run the AssemblyLine; with default parameters. In order to accommodate this situation, you need to do the following:

a. Edit the global.properties or solution.properties file to alter the settings of com.ibm.di.javacmd to refer to a batch file. (for example, com.ibm.di.javacmd=/opt/IBM/TDI/V6.1/myjava.bat)
b. Create a command file (the aforementioned /opt/IBM/TDI/V6.1/myjava.bat) containing the appropriate Java invocation command, like `"java" -Xms254m -Xmx1024M $*`

Now the CE will use the modified JVM invocation, with increased heap size.

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 the ″Required user privileges″ section.

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

   Note: If you run the 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 the ″Required Setup of the IBM Tivoli Directory Integrator″ section.

Compatibilty: Refer to the section on 26 on how this connector should be set up with the necessary libraries, and about incompatibilities with other Domino®/Lotus Notes® Connectors.

Domino Users Connector

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

- 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 2. 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 on

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 "useFTSearch" parameter:

- **"useFTSearch" = "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.
- **"useFTSearch" = "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 (i.e. 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 6.1: Password Synchronization Plug-ins Guide*.

Note: Domino Users Connector requires Lotus Notes to be release 6.0 or 6.5; and Lotus Domino Server version 6.0 and 6.5..

Deployment and connection to Domino server

Refer to the section, 26 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:

Domino Server IP Address
The IP Address 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.

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" (default)

Authentication Mechanism

This parameter specifies the authentication mechanism used by the Connector.

When "Internet pasword" authentication is selected, the **userName** Connector parameter must set to the Domino user's Short Name and the **password** Connector parameter must be set to the Domino user's Internet password.

When "Notes ID File" authentication is selected, the **userName** Connector parameter is ignored, because the Domino user to be used is unambiguously identified by the currently configured default Notes ID file. When "Notes ID File" authentication is selected, the **password** Connector parameter must be set to the password of the currently configured default Notes ID file.

For more information about the two supported authentication mechanism, please see the "Authentication" section.

Username

The user name used for log in or authentication to the Domino Server. Ignored if "Notes ID File" authentication is selected. See "Authentication" on page 43 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 43 for more details.

Use full-text search

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. Default value is " ".

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 to this formula:

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

which limits the search to user documents only. Default value is " ".

Detailed Log

If this field is checked, an additional log message is generated.

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

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

This authentication mechanism uses the currently configured default Notes ID file along with its password.

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 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_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_Server	String	No	The name of the server containing the user's mail file. If this attribute is missing, the value is obtained from the Connector's Domino Session object.

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_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_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.

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.
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_MinPassword Length	Integer/String	No	The REG_MinPassword Length 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_MinPassword Length value. If this attribute is missing, a default value of 0 is used.
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.

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 53 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 6.0 for AIX/Linux/Solaris

For Domino Users Connector with Domino Server 6.0 for AIX/Linux/Solaris, 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

"Lotus Notes Connector."

Lotus Notes Connector

The Lotus Notes Connector provides access to Lotus Domino databases.

The Lotus Notes Connector reads, writes and deletes records in any Notes database, and is therefore not limited to the Domino directory. Managing users in Lotus Notes requires modifying certificates, ACLs and mailboxes. This must be done manually or using the Config Editor. Users can be provisioned in and out of Notes by applying staging databases and integrating with Config Editor through Notes scripting.

Note: Lotus Notes Connector requires Lotus Notes to be release 5.0.8 or higher.

Known limitations

For 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–>Tools–>UserID** panel. The checkbox is labeled **Don't prompt for a password from other Notes-based programs. (Share this user ID password with these Notes add-ins)**. 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 26 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.

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:

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**.

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.

Configuration

The Connector needs the following parameters:

Hostname
: 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.

Session Type
: Can be one of **IIOP**, **LocalClient** or **LocalServer**. See "Session types" on page 57.

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.

Server The name of the server where **Database** is found. Leave blank to use the server you are connecting to (**Host Name**).

Database
The name of the database to use.

Document Selection
The selection used when iterating the data source. 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 field is checked, an additional log message is generated.

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.

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.

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" 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()));
```

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
This parameter 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.

Detailed Log
Turns on debug messages. This parameter is common to all TDI 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.

The DSMLv2 SOAP Connector supports Server mode.

Note: The DSMLv2 SOAP Server Connector is not designed as a replacement of the "DSMLv2 EventHandler" on page 199. The Connector provides a generic functionality for processing DSMLv2 requests.

Extended Operations

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
The TCP port on which the DSMLv2 SOAP Server Connector is listening.

Connection Backlog
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
This parameter determines if clients must provide HTTP basic authentication.

Auth Realm
The authentication realm sent to the client when requesting HTTP Basic authentication

Binary Attributes
This parameter 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
When this parameter is enabled, 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 TDI components.

Exchange Changelog Connector

The Exchange Changelog Connector is a specialized instance of the LDAP Connector. The Exchange Connector contains logic to poll Exchange for changed objects using the uSNChanged mechanism.

The Exchange Changelog Connector operates in Iterator mode.

Notes:

1. Microsoft dropped the extended support for Exchange 5.5 at the end of 2005. As TDI 6.1 does not support NT4, the Exchange Changelog Connector is *deprecated* along with the Exchange Changelog EventHandler in TDI 6.1.

2. The Exchange Changelog Connector connects to Microsoft Exchange Server 5.5; . Microsoft Exchange is usually used with Windows NT4. Users of Windows 2000 Server with Exchange 2000 and above would normally use Active Directory As a result the Exchange Changelog Connector is only applicable in Windows NT4 environment.
3. The Exchange 5.5 Service Pack 4 must be installed on the Exchange Server.

Behavior

When started, the Exchange Changelog Connector reads from the IBM Tivoli Directory Integrator User Property Store the USN values stored from the most recent Exchange Connector's session. The Exchange Connector first retrieves the newly added Exchange objects, then the modified and deleted Exchange objects. After an Exchange object is retrieved, it is parsed and its attributes and attribute values are copied to a new Entry object. This Entry object is then returned by the Exchange Connector. When there are no more changed objects to retrieve, the Exchange Connector cycles, waiting for a new change to happen. The **Sleep Interval** parameter specifies the number of seconds the Exchange Connector sleeps between successive polls when waiting for new changes. The Exchange Connector loops until either a new change is retrieved or the timeout expires. If the timeout expires, the Exchange Connector returns a **null** Entry, indicating that there are no more Entries to return. If a changed object is retrieved, it is processed, and the new Entry is returned by the Exchange Connector.

The Exchange Changelog Connector delivers changed Exchange 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. Exchange does not have a changelog, it only stamps an object when it is changed. Each object reported by the Exchange Connector represents the cumulative effect of all changes performed to that object. The Exchange Connector, however, recognizes the type of object change that must be performed on the replicated data source, and reports whether the object has to be added, modified or deleted in the replicated data source.

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

Using the Exchange Changelog Connector

The Exchange Changelog Connector adds the **changeType** attribute to every Entry returned. The possible values of the **changeType** attribute are **add**, **modify** and **delete**. These are used to represent new, changed and deleted objects respectively.

When an object is deleted in the Exchange Server, it is marked as **tombstone** and its "**Is-Deleted**" attribute is set to **true**. The object, however, is not moved to other container (like in the Active Directory Connector), and keeps its **distinguishedName** attribute. Exchange objects cannot be moved. As a result, the **distinguishedName** attribute never changes in Exchange and can be used safely for object identifier when synchronizing.

Note: Deleted objects in Exchange live for a configurable period of time (30 days by default), after which they are completely removed. To avoid missing deletions, perform incremental synchronizations more frequently.

It is possible for an object to be reported as modified, even though the object's contents have not changed since it was reported as a new object. This happens when the object has been created before the current Exchange Connector session has started and is later modified after the current Exchange Connector session has started but before the Exchange Connector has actually retrieved the object. In such a case the Exchange Connector retrieves the object's modified contents and reports it as a new object. After a short period of time (during the same session) the Exchange Connector reports the same object as modified and retrieves the same changed and already reported object's contents. Do not be concerned about this behavior since it neither hurts the synchronization process, nor has it any significant performance overhead.

It is also possible for an already deleted object to be reported as a new object and later reported as a deleted object during the same Exchange Connector session. This happens when the object has been created before the current Exchange Connector session has started and is later deleted after the current Exchange Connector session has started, but before the Exchange Connector has actually retrieved it. In this case the Exchange Connector retrieves the object's deleted state and reports it as a new object. After a short period of time (during the same session) the Connector reports the same object as deleted. The addition of an already deleted object can be recognized by checking for the ″**Is-Deleted**″ attribute. If the object is present and has a value of **true**, the object is already deleted.

The Exchange Changelog Connector can be interrupted 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 Connector is started it successfully continues the synchronization from the point the Exchange Changelog Connector was interrupted.

The Exchange Changelog Connector supports the IBM Tivoli Directory Integrator 6.1 Checkpoint/Restart functionality.

The Is-Deleted attribute in Exchange

By default, Exchange does not expose the **Is-Deleted** object attribute through LDAP. However, the **Is-Deleted** attribute can be used in LDAP search queries when it is not returned as part of the returned Exchange object. That is why the Exchange Connector properly detects the type of change even if the **Is-Deleted** attribute is not visible through LDAP. If the **Is-Deleted** attribute is visible through LDAP, then the Exchange Connector can retrieve changes quicker if you set the ″**Is-Deleted**″ **Attribute Visible** parameter to **TRUE**. The Connector uses the returned **Is-Deleted** attribute and completes the work faster.

Note: If the server does expose the **Is-Deleted** attribute, but ″**Is-Deleted**″ **Attribute Visible** is set to **false**, then the Connector still works properly, but you can accelerate the Exchange Connector by setting ″**Is-Deleted**″ **Attribute Visible** to **true**. If the server

does not expose the **Is-Deleted** attribute, but ″**Is-Deleted**″ **Attribute Visible** is set to **true**, then the Connector cannot distinguish between modify and delete and reports modify for both modify and delete operations.

To expose the **Is-Deleted** attribute you must obtain Exchange server administrator privileges and do the following:

1. Start the Exchange Administration program **admin.exe** in RAW mode (**admin /r**).
2. Select **View->Raw Directory** from the menu.
3. Select **Schema** from the window on the left.
4. Double-click **Is-Deleted** from the window on the right.
5. A message box is displayed, informing you that only raw properties are displayed. This message box asks if you want to view these raw properties. Click **Yes**.
6. Select the **Heuristics** property. You can see the current property value on the right. You can also edit the property value.
7. To decide what value to set the Heuristics property to, please refer to the following. The heuristic property is a bit mask, which is interpreted as follows:

Bit 0

- **0**: Replicate between sites
- **1**: Do not replicate between sites

Bit 1

- **0**: Attribute is not visible through LDAP
- **1**: Attribute is visible to anonymous and authenticated LDAP clients

Bit 2

- **0**: Attribute is not accessible by authenticated clients
- **1**: Attribute is accessible to authenticated clients but not anonymous clients

Bit 3

- **0**: Attribute is not an operational attribute
- **1**: Attribute is an operational attribute

Bit 4

- **0**: Attribute is not visible in Config Editor (Attributes page of DS Site Configuration object)
- **1**: Attribute is visible in Config Editor (Attributes page of DS Site Configuration object)

By taking note of heuristics, you can determine the visibility of particular attributes. For example, a heuristic value of **3** means the attribute is not replicated between sites and is visible by anonymous LDAP clients. A heuristic value of **11** means the attribute is an operational attribute and is visible to authenticated LDAP clients.

As can be seen from the table, bit **1** of the heuristics value determines whether an object is visible through LDAP. That is why you need to set bit **1** to make the attribute visible through LDAP.

Another important property of an Exchange object or an Exchange object attribute is the Description property. It determines the LDAP name of the attribute or object.

Note: Changing the LDAP name might cause interoperability problems.

Accessing the USN synchronization values in the User Property Store

The state of synchronization at any time is represented by 4 USN numbers:

- START_USN
- END_USN
- CURRENT_USN_CREATED
- CURRENT_USN_CHANGED

These values are packed and stored in the User Property Store. Do not change these values manually. You might want to archive the numbers corresponding to a certain stage of synchronization and later use these numbers to replay synchronization from that stage.

The following script code can be used in the IBM Tivoli Directory Integrator to get USN values stored in the User Property Store:

```
// Retrieve USN values from User Property Store
var usn = system.getPersistentObject("exchange_sync");
var startUsn = usn.getString("START_USN");
var endUsn = usn.getString("END_USN");
var currentUsnCreated = usn.getString("CURRENT_USN_CREATED");
var currentUsnChanged = usn.getString("CURRENT_USN_CHANGED");

main.logmsg("START_USN: " + startUsn);
main.logmsg("END_USN: " + endUsn);
main.logmsg("CURRENT_USN_CREATED: " + currentUsnCreated);
main.logmsg("CURRENT_USN_CHANGED: " + currentUsnChanged);
```

"exchange_sync" is the name of a parameter already stored in the User Property Store. The **Iterator State Key** parameter specifies this value. The above example just dumps the values to the screen, but you may do whatever you want with them - for example save them in a file and backup this file.

The next script code example shows how the USN values can be stored in the User Property Store:

```
// Store USN values in the User Property Store
var usn = system.newEntry();
usn.setAttribute("START_USN", startUsn);
usn.setAttribute("END_USN", endUsn);
usn.setAttribute("CURRENT_USN_CREATED", currentUsnCreated);
usn.setAttribute("CURRENT_USN_CHANGED", currentUsnChanged);
system.setPersistentObject("exchange_sync", usn);
```

This code assumes that the variables *startUsn*, *endUsn*, *currentUsnCreated* and *currentUsnChanged* contain the USN numbers as strings. This example saves the USN values under the `"exchange_sync"` parameter, so `exchange_sync` must be specified in the **Iterator State Key** parameter to continue synchronization from the desired point.

Accessing the runtime Connector's USN synchronization values

The Exchange Changelog Connector provides the following public methods for access to its current USN values:

public Entry getUsnValues ();

Returns an Entry object with the following attributes:

- START_USN
- END_USN
- CURRENT_USN_CREATED
- CURRENT_USN_CHANGED

The value of each of the attributes is of type java.lang.Integer and represents the corresponding Connector's USN value.

public void setUsnValues (Entry usnEntry);

Sets the Exchange Connector's current USN synchronization values to the values specified in the **usnEntry** parameter. The structure of the **usnEntry** parameter must be the same as the structure of the Entry returned by getUsnValues(). The values of the **usnEntry** attributes must be either **java.lang.Integer** or the String representations of the corresponding numbers.

Note: Be careful when changing the USN values at runtime. Specifying inconsistent values can result in improper synchronization.

Configuration

The Connector needs the following parameters:

LDAP URL

The LDAP URL of the Exchange server 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

The distinguished name used for authentication to the service. For example, **cn=administrator,ou=domain_name,o=organization_name**.

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

Login password

The credentials (password).

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

Authentication Method

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 Exchange Server.

LDAP Search Base

The specified Exchange sub-tree which is polled for changes. For example, **cn=recipients,ou=domain_name,o=organization_name**.

USN Filename

Specifies the name of the text file where the most recent Connector session USN values are stored. If the file does not exist or is not in the correct format, the Connector performs a full synchronization with Exchange. The file must not be read-only, as the Connector writes the updated USN values each time it retrieves an Entry.

Note: The use of a separate USN file by means of this parameter is deprecated. Use the **Iterator State Key** parameter instead.

If you change the search base or the server or both, you might want to change or edit the USN file as well, because the stored USN values might be invalid in regard to the new search base or server.

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 a start USN number from which synchronization must start. **EOD** means report only changes that happen after the Connector has started.

"Is-Deleted" Attribute visible

Specifies whether the Exchange server exposes the **Is-Deleted** object attribute through LDAP.

Note: If the server does expose the **Is-Deleted** attribute, but "**Is-Deleted**" **Attribute visible** is set to **false**, then the Connector still works properly. You can accelerate the Connector by setting "**Is-Deleted**" **Attribute visible** to **true**. If the server does not expose the **Is-Deleted** attribute, but "**Is-Deleted**" **Attribute**

visible is set to **true**, then the Connector cannot tell the difference between a modified object and a deleted object and reports all deletions as modify change operations.

Timeout

Specifies the number of seconds the Connector searches for a new change. Regardless of the value of this parameter, the Connector does not timeout until it has scanned every change that has happened before the current Connector session has started.

If this parameter is **0**, then the Connector waits indefinitely for the next change.

When the Connector times out, it returns an empty (**null**) Entry, thus indicating that there are no more Entries to return.

Sleep Interval

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

Detailed Log

If this field is checked, an additional log message is generated.

Comment

Your comments here.

Migration

For IBM Tivoli Directory Integrator 6.1, the Exchange Changelog Connector stores the USN synchronization values into the User Property Store of the IBM Tivoli Directory Integrator. Older versions (pre-6.0) of the Exchange Changelog Connector store the USN values in a text file.

Do the following to migrate a pre-6.0 IBM Tivoli Directory Integrator configuration:

1. Open the pre-6.0 configuration and set a value for the **Iterator State Store** parameter.
2. Run the AssemblyLine. The Exchange Connector reads the start USN values from the text file, but stores the updated USN values both in the text file and in the User Property Store.
3. Stop the AssemblyLine.
4. Clear the value of the **USN Filename** parameter and save the configuration. The Connector reads and stores the USN synchronization values in the User Property Store and no text file is used.

See Also

"LDAP Connector" on page 118
"Active Directory Changelog (v.2) Connector" on page 7,
"Netscape/iPlanet Changelog Connector" on page 139,
"IBM Directory Server Changelog Connector" on page 87.
"z/OS Changelog Connector" on page 187.

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. 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 field is checked, an additional log message is 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 182.

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.
3. The FTP Client in AddOnly mode with Save Connector Restart Info does not support Checkpoint/Restart.

Configuration

The Connector needs the following parameters:

FTP Hostname
The hostname 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. Specify **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.

Detailed Log
If this field is checked, an additional log message is 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 bottom-right **Inherit From:** button.

See also

"The FTP object" on page 358,
"URL Connector" on page 182,
"Old HTTP Client Connector" on page 78,
"Old HTTP Server Connector" on page 86.

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 TDI truststore, and configure the truststore in global.properties or solution.properties.

Note: The HTTP Client Connector does not support the Advanced Link Criteria (see "Advanced link criteria" in *IBM Tivoli Directory Integrator 6.1: 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.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.

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.

Detailed Log
If this field is checked, an additional log message is generated.

You select a Parser from the Parser pane; select the parser by clicking the bottom-right **Inherit From:** 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 182,
"HTTP Server Connector" on page 81,
"HTTP Parser" on page 242.

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.

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 6.1: 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 field is checked, an additional log message is 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 182,
"Old HTTP Server Connector" on page 86,
"HTTP Client Connector" on page 75.

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".

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 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 does the following:

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 the Connector creates a new instance of itself and this new instance is handed over to the AssemblyLine that spawns a new AssemblyLine thread for that Connector instance; this design feature provides the ability of processing 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 of HTTP clients accessing this connector over the network.

There are two different ways to implement the HTTP Basic Authentication with the HTTP Server Connector:

1. **Using an Authentication Connector**

   This is a mechanism for backward compatibility with the HTTP EventHandler. A connector parameter **Auth Connector** specifies a TDI Connector that will be used to lookup the username and password for the HTTP Basic Authentication data:

   - 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 his 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 on your own through 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 as chunks.

When chunked encoding is used the user is responsible to call 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 enabled by checking the check box (default), the TCP connection properties are accessed through the getProperty() method of the `work` Entry object. If unchecked, the TCP connection properties appear as Entry Attributes.

HTTP Headers as Properties
If enabled by checking the check box (default), all HTTP headers are accessible using the getProperty() method of the work Entry object. If unchecked, all HTTP headers appear as Entry Attributes.

HTTP Basic Authentication
If enabled (which 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 (e.g. 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 82.

Use SSL
If this parameter is set to true/checked, then the Connector will require clients to use SSL; non-SSL connection requests will fail. The default is off.

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

Require Client Authentication
If enabled (which 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 which can be matched to the configured TDI 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 83.

Detailed Log
If this field is checked, an additional log message is 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 182,
“HTTP Client Connector” on page 75,
“HTTP Parser” on page 242.

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.

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 does the following:

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 field is checked, an additional log message is generated.

Parser The name of a Parser to handle the contents of **post** requests.

See also

"URL Connector" on page 182,
"HTTP Server Connector" on page 81.

IBM Directory Server Changelog Connector

The IBM Directory Server Changelog Connector is a specialized instance of the LDAP Connector. The IBM Directory Server Changelog Connector contains logic to iterate the Changelog. It returns various attributes, including the **changes** attribute. The Connector returns **changes** as something that looks like a standard attribute, but it is in fact of the Entry class.

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 operate as in TDI 6.0 and 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.

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
The authentication method. Possible values are:

- CRAM-MD5 (use the CRAM-MD5 (RFC-2195) SASL mechanism).
- none (use no authentication (**anonymous**)).
- simple (use weak authentication (cleartext password)).
- If not specified, default (simple) is used. If **Login username** and **Login password** are blank, then **anonymous** is used.

Use SSL
If Use SSL is **true** (i.e., 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
This parameter 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 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.

State Key Persistence
This 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
This 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
This updates the System Store with the change log number when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual
This 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.

Use Notifications
This specifies whether to use notification when waiting for new changes in IBM Directory Server. If enabled, the Connector will not sleep or timeout but instead wait for a Notification event from the IBM Directory Server.

Batch retrieval
This 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, i.e. 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.

See also

"LDAP Connector" on page 118,
"Active Directory Changelog (v.2) Connector" on page 7,
"Exchange Changelog Connector" on page 65
"Netscape/iPlanet Changelog Connector" on page 139,
"z/OS Changelog Connector" on page 187.

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 key store/trust store) 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 TIM Agent's certificate into into the TDI Trust store.

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 trust store 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 Entries; 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. 1. Make sure you are using the latest version of the Agent.
2. 2. Try the work-around described above.
3. 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.

JDBC Connector

The JDBC Connector provides 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 TDI 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.

Notes:

1. The JDBC Connector is supported on Microsoft Access 2000 and later versions. This is for a specific JDBC Connector for Access, not using the ODBC bridge. However, both approaches can be used. Use the ODBC bridge to access older versions of Access, just like you use the bridge to access Excel or other source-exposed as ODBC.
2. ODBC, by means of the standard Java2 JDBC-ODBC bridge, is supported on Windows systems only, and performance is likely to be sub-optimal compared to a dedicated, native driver. Commercial ODBC/JDBC bridges are available. If you need an ODBC/JDBC bridge, consider purchasing a commercially available bridge.
3. For the Oracle database system, the supported version of the driver is 92. For more information, refer to Oracle's *JDBC Developer Guide and Reference Release 9.2*.
4. Information about the JDBC driver for IBM DB2® is available online; a starting point and example for configuration purposes is http://publib.boulder.ibm.com/infocenter/db2help/topic/com.ibm.db2.udb.doc/ad/t0006330.htm, in the section on Administering XML extender in the DB2 Administering documentation. The version of `db2jcc.jar` bundled with TDI is the driver that is part of CloudScape – this driver may or may not suit your purpose. This driver does not need further licensing; however, the driver that is part of the IBM DB2 Connect™ product does.
5. If you are running DB2 on a zOS 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.

The JDBC Connector also provides multi-line input fields for the SELECT, INSERT, UPDATE and DELETE statements used by the JDBC connector. 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 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 are also handled and set during connection initialization.

The JDBC connector builds SQL statements internally using the SqlMap. 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 will be 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. In Delta mode the Connector will use the SQL statements for add, modify or delete, corresponding to what the operation is. (The same for as for Update, but that one allows you to only use add or modify).

Taking advantage of PreparedStatements

The JDBC connector uses *PreparedStatements* 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 is available in the JDBC connector's configuration. The config parameter is a Boolean flag called usePreparedStatement (configured via a checkbox called **Use Prepared Statements** in the connector config screen) which indicates whether the JDBC connector should use PreparedStatements. If this is set (the default) 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 TDI solution developer to handle situations when there are problems due to use of PreparedStatements.

The findEntry, putEntry, deleteEntry and the modEntry methods of the JDBC connector checks 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.

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 four operations where the templates can be used. These operations are:

Table 3.

Operation	Description	Mode(s)
SELECT	Used in Iterator mode (no search criteria)	Iterator
INSERT	Used when adding an entry to the data source	Update, AddOnly
UPDATE	Used when modifying an existing entry in the data source.	Update
DELETE	Used when deleting an existing entry in the data source.	Delete
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 (e.g. null or empty) the JDBC connector will behave as before.

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 (e.g. mc, config, work, Connector) as well as the JDBC schema for the table configured for the connector and a few other convenience objects.

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 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 also not to LOOKUP) statements. This value is a comma separated 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}
```

e.g., *SELECT a,b,c from TABLE-A WHERE a > 1 AND b = 2*

Table 4. Information available for different statements

	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

Table 4. Information available for different statements (continued)

	SELECT	LOOKUP	INSERT	DELETE	UPDATE
link	no	yes	no	yes	yes
whereClause	no	yes	no	yes	yes

Configuration

The Connector needs the following parameters:

JDBC URL
See documentation for your JDBC provider. It is called JDBC URL in the IBM Tivoli Directory Integrator Config Editor. For IBM DB2, a typical URL would be `jdbc:db2://hostname:port/dbname` , Oracle URLs for their ″thin driver″ look like `jdbc:oracle:thin:@hostname:1521:SID` .

JDBC Driver
The JDBC driver class name. The default value of `sun.jdbc.odbc.JdbcOdbcDriver` addresses the JDBC-ODBC bridge, which is not recommended for production use. For IBM DB2, the type 2 or type 4 driver you would typically specify is `com.ibm.db2.jcc.DB2Driver` , whereas for Oracle, this would be `oracle.jdbc.driver.OracleDriver` .

Username
Only the tables available to this user are shown.

Password
The password 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 TDI 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 and objects grouped under one identifier (username). For this particular parameter in this particular Connector, we mean to signify 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...
Press this button to bring up a list of valid table names that you can select from to enter into the **Table Name** field. This only works if the underlying database supports this; e.g. Microsoft Access using ODBC does not.

Return null values
If this parameter is set to **true**, then **NULL** values are returned as empty attributes (for example, empty value set). If set to **false**, then the attribute is not part of the entry returned.

Commit
Controls when database transactions are committed. Options are:

- **After every database operation** (default)
- **After every database operation (Including Selects)**
- **On Connector close**
- **Manual**

Manual means user must call the *commit()* method of the JDBC Connector—or *rollback()*, as appropriate.

Note: The option **After every database operation (Including Selects)** has been provided for those databases which lock database tables in transactions even when they only have been Selected for read operations (notably DB2).

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.

SQL Lookup
The custom SQL statement to use for lookups (used in Lookup and Delete modes).

SQL Insert
The custom SQL statement to use when inserting into the database, using AddOnly mode.

SQL Update
The custom SQL statement to use when updating the database, using Update mode.

SQL Delete
The custom SQL statement to use when using Delete 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. E.g.: 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.

Use Prepared Statements
This value of this check box determines whether to use PreparedStatement or Statement. If this is selected the PreparedStatement will be used by the JDBC connector else Statement will be used. The default is checked, i.e. "true", meaning try to pre-compile SQL statements, fall back to normal.

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

The operand **Equal** is translated to the equal sign (=), while the **Contains**, **Start With** and **End With** operators are mapped to the **like** operator.

Other

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*, e.g. `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 does not interfere with the normal flow of entries and attribute mappings for the Connector.

Timestamp

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 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 TDI 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). In case, for example, 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").getTi
  ```
- 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.

Calling Stored Procedures

he JDBC Connectors "connection" property 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.connector.connection;
```

will give you access to the JDBC Connection object (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();
    // TDI 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 character 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 ignore this section unless you are curious about the internals.

For a database, the Connector uses prepared statements or dynamic query depending on the situations:

- If the Connector gets the schema definition from the database, it uses prepared statements.
- Otherwise, the Connector creates a dynamic SQL query.

On Multiple Entries

See Appendix B, “AssemblyLine and Connector mode flowcharts,” on page 363 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.

ODBC–specifying database paths directly

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:

MS Access is installed

Open the ODBC data source control panel a 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
```

MS Access is not installed

If MS Access is not installed, and you are on a Windows system, use the following:

```
jdbc:odbc:Driver={MS Access Driver
 (*.mdb)};dbq=C:\Documents and Settings\<username>\My Documents\mydb.mdb
```

Alternatively, use the Windows Named System DSN utility, available under Administrative Tools > ODBC Data Sources. Once you define a System DSN, use a jdbcSource parameter like the following:

```
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.

JMS Connector

Introduction

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 "**IBMMQ 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 6.1: 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.

When deploying the JMS Connector in conjunction with the Checkpoint/Restart functionality available with IBM Tivoli Directory Integrator, careful thought must be given to the

acknowledgement of received messages (see ″Checkpoint/Restart — Saving and storing AssemblyLine state information″ in *IBM Tivoli Directory Integrator 6.1: Users Guide*). 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 Checkpoint/Restart 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 Checkpoint/Restart 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 (i.e., 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.

Add Only 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. As opposed to JMS headers, 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.

  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.

Use SSL Connection
: Enables 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.

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 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 == forever**

JMS Server Type
Select the JMS server type.

Specific Driver Attributes
Thes 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 6.1: Administrator Guide* and the "System Queue Connector" on page 146.

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 field is checked, an additional log message is 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 *root_directory*/examples/jms directory of your IBM Tivoli Directory Integrator installation.

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 (1) taken from the WebSphere MQ installation, (2) placed under the "<tdi_root_folder\jars\3rdparty\IBM>" folder and then (3) added to the TDI class path:

- com.ibm.mqjms.jar
- com.ibm.mq.jar
- jms.jar
- connector.jar

In order to enable a Secure Socket Layer (SSL) session, you must first configure a channel on your MQ server. Detailed instructions on how to perform these tasks are included in a Technical Journal Article, "Configuring SSL Connections between JMS Clients and the WebSphere MQ JMS Provider", http://www7b.software.ibm.com/wsdd/techjournal/0211_yusuf/yusuf.html, Dr Kareem Yusef, November 2002. This article also provides detailed instructions on obtaining and managing certificates required to run an SSL test. The version of IBM WebSphere MQ should be at least v5.3 service level 5.3.0.4 (i.e, fix pack CSD04 installed prior to attempting SSL configurations. In the TDI Properties store, specify the trustStore and keyStore settings prior to starting IBM Tivoli Directory Integrator. For example:

```
javax.net.ssl.trustStore=d:\\jdk141\\jre\\lib\\security\\cacerts
javax.net.ssl.trustStorePassword=
javax.net.ssl.trustStoreType=
```

```
javax.net.ssl.keyStore=C:\\Program Files\\IBM\\WebSphere MQ\\Java\\bin\\jmskeystore
javax.net.ssl.keyStorePassword=changeit
javax.net.ssl.keyStoreType=jks
```

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 TDI is installed.

JMX Connector

The JMX Connector uses the JMX 1.2 and JMX Remote API 1.0 specifications. It nly 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 stores the notification details) to the AssemblyLine.

Notifications which are received in-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 wild-card 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 as per the JMX Remote API 1.0 specification.

The available values (drop-down list) for this parameter are "remote" and "local".

"local" means that the Connector will only listen for notifications emitted by MBeans registered with an MBeanServer in the local Java Virtual Machine.

"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 emitted 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
This parameter specifies whether the Connector will register with all available MBeans (checked/true) or only with the ones specified in the **MBeans to listen to** Connector parameter (unchecked/false). This parameter is checked by default.

MBeans to listen to
This parameter 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 (i.e. the list is empty) notifications emitted by any MBean will be reported. If at least one MBean name is specified, then only notifications emitted 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.

This parameter 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
Check this to enable more log messages.

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.

JNDI Connector

The JNDI Connector provides access to a variety of JNDI services. To reach a specific system you must install the JNDI driver for that system. 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 `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.

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, e.g. ldap://host for the LDAP driver.

Authentication Method
The type of JNDI authentication to be used; choose from the drop-down list. Choices are:

- **Anonymous** (use no authentication)
- **Simple** (use weak authentication (cleartext password))
- **CRAM-MD5** (use CRAM-MD5 (RFC-2195))
- **SASL** (use SASL)

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.

Provider Params
A list of extra provider parameters you want to pass to the provider. Specify each *parameter:value* on a separate line.

Detailed Log
If this field is checked, an additional log message is generated.

Setting the Modify operation

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 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)`

See also

"LDAP Connector,"
"MQe Initialization" on page 148.

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 6.1: 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://ldap.akbkhome.com/objectclass/person.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:

```
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 124.
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 **Connector Flags** does not contain **deleteEmptyStrings**, then empty strings are passed as permitted values to the directory server. Most servers interpret a REPLACE request with an empty string the same as removing the attribute altogether. If you want to control this behavior, you can call a function in your **Before Update** hook to modify the entry as in:

```
removeBlanks (work);

function removeBlanks (entry) {
 var list = entry.getAttributeNames();
 for (i = 0; i < list.length; i++) {
  if (entry.getString(list[i]) == "") {
   entry.removeAttribute (list[i]);
  }
 }
}
```

Configuration

The Connector needs the following parameters:

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 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
: 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 searchbase.

Time Limit
Searching for Entries must take no more than this number of seconds. **0 = no limit**.

Size Limit
A search or iteration must return no more than this number of Entries. **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.

Comment
Your comments here.

Authentication Method
Type of LDAP authentication. Can be one of the following:

- Simple (using **Login username** and **Login password**). Treated as anonymous if **Login username** and **Login password** not provided).
- MD5-CRAM.
- SASL (parameters for this type of authentication will need to be specified using the **Extra Provider Parameters** option.)
- Anonymous (treated as Simple if **Login username** and **Login password** are supplied).

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.

LDAP Trace File

Trace LDAP BER packets to file.

Sort Attribute
A new 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.

Attention: This parameter causes the Connector to use the Virtual List View LDAP control. In TDI 6.1 this LDAP control and hence this configuration parameter is deprecated. See the "Virtual List View Control" section for more details. LDAP paging should be used instead of the LDAP Virtual List View - see the **Page Size** configuration parameter for more information.

Simulate Rename
If the server does not support rename, simulate it with **delete** or **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 **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.

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

Virtual List View Control

In TDI 6.1 this LDAP control and hence the **Virtual List View Page Size** configuration parameter is deprecated. Newly created TDI 6.1 configurations as well as pre-TDI 6.1 configurations which didn't use this param (i.e. used the default value 0) will have this attribute disabled in the GUI. Any pre-TDI 6.1 configurations which did use this parameter will still use it with 6.1 and the parameter will be enabled in the GUI. Note: In order to use the Virtual List View Control in TDI 6.1, 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_folder>\jars" folder before starting TDI. 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 a couple of 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 6.1: 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.

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>Low Level API** 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 a command similar to the following:

```
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 also

"JNDI Connector" on page 114,
"Active Directory Changelog (v.2) Connector" on page 7,
"Exchange Changelog Connector" on page 65
"Netscape/iPlanet Changelog Connector" on page 139,
"IBM Directory Server Changelog Connector" on page 87
"z/OS Changelog Connector" on page 187.

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 6.1: 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 EventHandler 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 118

Lotus Notes Connector

See "Lotus Notes Connector" on page 56.

Mailbox Connector

This Mailbox Connector provides access to internet mailboxes (POP3 or IMAP). The Mailbox Connector can be used in Iterator, Lookup and Delete modes. The Mailbox Connector uses predefined attribute names for the most used headers. If you need more than this use the **mail.message** property to retrieve the native message object.

When the Mailbox Connector is used in Lookup or Delete mode the only searchable headers are:

- mail.from
- mail.to
- mail.cc
- mail.subject
- mail.messageid
- mail.messagenumber

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.

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 6.1: 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
This parameter 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.

Poll Interval (seconds)
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, i.e. the Connector polls the server for new mail messages.

This parameter specifies the amount of seconds that the Connector will sleep before polling the mail 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 field is checked, an additional log message is generated.

Predefined properties and attributes

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.

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 Mailbox EventHandler.

mail.session
he 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 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 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 Mailbox EventHandler.

See also

"Mailbox EventHandler" on page 216.

Memory Queue Connector

The 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 FC" on page 279 (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 could contain embedded logic that would trigger whenever buffer is x% full/empty/data available.

This Connector supports AddOnly and Iterator modes only.

Notes:

1. Because of the non-persistent nature of this Connector, you should rather use the "System Queue Connector" on page 146 instead, because that Connector relies in the underlying Java Messaging Service (JMS) functionality with persistent object storage.
2. By default, if the Memory Queue connector in Iterator mode starts reading from the queue but it doesn't exists, it will create it. If you don't want this behavior, you would need to set

the system property **tdi.memq.create.queue.default=false**, in this case TDI 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 the a MemQueue pipe created by the MemQueue Connector will be terminated when the Connector closes.

Notes:

1. Because of the non-persistent nature of this Connector, you should rather use the "System Queue Connector" on page 146 instead, because that Connector relies in the underlying Java Messaging Service (JMS) functionality with persistent object storage.
2. By default, if the Memory Queue connector in Iterator mode starts reading from the queue but it doesn't exists, it will create it. If you don't want this behavior, you would need to set the system property **tdi.memq.create.queue.default=false**, in this case TDI will behave like previous versions; this implies that when the queue does not exists, an exception is thrown.

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 optional persistence using the System Store, when a threshold value is reached, which is a function of the runtime memory available.

The Memory queue buffer should be constructed using the MemoryBufferQFactory.

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 Collection of objects, *pagesize* specified by the user, which does chunking of data before it is written to the system store, to optimize database writes. This is done by dividing the queue into pages and reading/writing one complete page to the system store so as to minimize the number of database read/write operations. This option is used only used when paging is use; 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 gobal 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

With Paging
The Memory queue Buffer is a queue of pages containing objects. When a particular threshold is reached, 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.

Without paging
The Memory queue Buffer is a queue of objects. When a particular watermark (specified by the user) is reached, the input is either blocked or it fails (can result in loss of data).

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.

Watermark
The Watermark parameter is either:

- With **Paging Off** (next parameter), it is the maximum queue size. When this size is reached, the **Blocking Add Parameter** determines if the read waits or fails since the queue is full.
- With **Paging On**, it 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.

Enable paging using system store
Check as to whether or not queue uses paging support using system store.

Page Size
Number of entries in one page.

Database name
System Store database name.

Username
Username to connect to the System Store database.

Password
Password to use when signing on to the database.

Table name
Database table name to be used.

Blocking add operation
Option to block or fail (and always log that data is lost) while adding, if queue is full and no paging used.

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 356. This Connector has an additional method, the getDataBuffer() method.

Configuration

Detailed Log
If this field is checked, an additional log message is generated.

Parser The name of a Parser to format the output or parse the input.

MQe Password Store Connector

Notes:

1. See *MQ Everyplace Password Store Installation and Setup* (*pwsync_install_directory*\IDS\ readme_mqepwstore_ismp.htm) for more information.In addition, more information about installing and configuring the IBM Password Synchronization Plug-ins, please see the *IBM Tivoli Directory Integrator 6.1: Password Synchronization Plug-ins Guide*.
2. IBM Tivoli Directory Integrator 6.1 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 and/or application submitting a change password request to the MQe Password Store Connector.
3. The MQe Password Store Connector receives password update messages from an MQe QueueManager. See also the section on "MQe Initialization" on page 148 in the documentation of the "System Queue Connector" on page 146 on how these two connectors utilize MQe on the same JVM.
4. MQ Everyplace does not support IP Version 6 addressing; as a consequence, the MQe Password Store Connector can only reach MQe using traditional IPv4 addresses.

The MQe Password Store Connector supports Iterator mode only.

The following is the MQe Password Store Connector workflow:

1. The MQe Password Store Connector requests a message from a predefined queue on its local MQe QueueManager using the JMS interface.
2. The retrieved message is 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 MQe Password Store Connector does the following:

- Get a reference to the MQe QueueManager.
- Initiates a connection to the Storage Component and notifies the Storage Component that the MQe Password Store Connector QueueManager is up.

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.

MQe Password Store Connector Entry structure

The MQe Password Store Connector constructs IBM Tivoli Directory Integrator Entry objects with the following fixed attribute structure:

UserID
Contains a single string value.

OperationType
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.

Storage notification server
Specify in a *host*:*port* format the Storage Component server that listens for notifications from the MQe 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.

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 MQe 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).

Detailed Log
Check this field for more detailed log messages.

See Also

"System Queue Connector" on page 146,
The chapter on System Queue in the *IBM Tivoli Directory Integrator 6.1: Administrator Guide*.

Netscape/iPlanet Changelog Connector

The iPlanet Changelog Connector is a specialized instance of the LDAP Connector.

In 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 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 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.

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
The authentication method. Possible values are:

- CRAM-MD5 (use the CRAM-MD5 (RFC-2195) SASL mechanism).
- none (use no authentication (**anonymous**)).
- simple (use weak authentication (cleartext password)).
- If not specified, default (simple) is used. If **Login username** and **Login password** are blank, then **anonymous** is used.

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 Base
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
This parameter 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 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.

State Key Persistence

This 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

This updates the System Store when you read an entry from the iPlanet Directory Server change log, before you continue with the rest of the AssemblyLine.

End of cycle

This updates the System Store with the change log number when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual

This 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.

Delivery Mode

Specifies whether to use changelog or notifications entries. If the LDAP Server doesn't maintain a changelog, **Realtime** is only applicable option. The default is **Changelog**.

Use Notifications

This specifies whether to use notification when waiting for new changes in iPlanet Directory Server. If enabled, the Connector will not sleep or timeout but instead wait for a Notification event from the iPlanet Directory Server.

Batch retrieval

This 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 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, i.e. 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.

See also

"LDAP Connector" on page 118,
"Active Directory Changelog (v.2) Connector" on page 7,,
"Exchange Changelog Connector" on page 65
"IBM Directory Server Changelog Connector" on page 87
"z/OS Changelog Connector" on page 187.

Server Notifications Connector

The Server Notifications Connector is an interface to the IBM Tivoli Directory Integrator (TDI) notification system. It listens for and reports, as well as emits, Server API notifications. The Connector provides the ability to monitor various processes taking place in the TDI Server, such as AssemblyLine stop and start processes, as well as emit custom server notifications.

The Server Notifications Connector supports the Iterator and AddOnly modes:

- In Iterator mode, when the Server API notification system emits an event, the event is acquired by the Server Notifications Connector and buffered in a queue internal to the Connector. On the next getNextEntry() method the Connector returns the next event buffered in the internal Connector queue.
- In AddOnly mode, the Connector is capable of emitting custom Server API notifications, which can be handled by TDI components which have registered to listen for these notifications through the Server API.

Encryption and Cryptography

The Server Notifications Connector provides the option to use Secure Sockets Layer (SSL) when the connection type is set to `remote`. If the remote TDI server accepts SSL connections only, the Server Notifications Connector automatically establishes an SSL connection provided that a trust store on the local TDI 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 TDI Server is needed because when the remote TDI server fires a notification a new SSL connection to the local TDI Server is created and in order for this new SSL connection session to be established the local TDI Server must trust (through its trust store) the remote TDI 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 TDI 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 TDI server.

User name and Password Authentication

The Server Notifications Connector is capable of using the Server API user name and password authentication mechanism. The desired user name and password can be set as a Connector parameter, in which case the Connector will use the Server API user name and password authentication mechanism. 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 an SSL client certificate to authenticate to the remote TDI Server.

Configuration

The Server Notifications Connector uses the following parameters:

connectionType
: This parameter 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 TDI Server system and register for and emit notifications in the Java Virtual Machine of that remote system.

url
: This is the Remote Method Invocation (RMI) URL used to connect to the remote TDI 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
: This parameter specifies the user name the Connector uses to authenticate to the TDI server. This parameter is only taken into account if the `connectionType` parameter is set to `remote`.

password
: Specifies the password the Connector uses to authenticate to the TDI server. This parameter is only taken into account if the `connectionType` parameter is set to `remote`.

configInstanceId
: This parameter 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 which have the Config Instance ID. This parameter is only taken into account if the Connector mode is Iterator.

notificationId
: This parameter 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
This parameter 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.

di_all This parameter specifies if 'di.*' notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

di_ci_all
This parameter specifies if 'di.ci.*' notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

di_ci_start
This parameter 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.

di_ci_stop
This parameter is only taken into account if the Connector mode is Iterator. This parameter specifies if 'di.ci.stop' notifications will be received by the Connector.

di_ci_file_updated
This parameter 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.

di_al_all
This parameter specifies if 'di.al.*' notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

di_al_start
This parameter 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.

di_al_stop
This parameter 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.

di_eh_al
This parameter specifies if 'di.eh.*' notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

di_eh_start
This parameter specifies if 'di.eh.start' notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

di_eh_stop
This parameter specifies if 'di.eh.stop' notifications will be received by the Connector. This parameter is only taken into account if the Connector mode is Iterator.

di_server_stop
This parameter 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.

hasCustomNotifications
This parameter specifies whether the Connector will receive any additional or custom notifications. If it is checked the additional/custom notifications can be specified in the "customNotifications" Connector parameter. This parameter is only taken into account if the Connector mode is Iterator.

customNotifications
This parameter specifies the notification types of additional/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 `hasCustomNotifications` parameter is true. This parameter is only taken into account if the Connector mode is Iterator.

Debug
This parameter turns on debug messages This parameter is globally defined for all TDI components.

System Properties Connector

This connector deals with "*property=value*" definitions, as in the configuration files `global.properties` and `solution.properties`, as well as other properties collections available in IBM Tivoli Directory Integrator Property Store.

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 supports Iterator, AddOnly, Update, Lookup and Delete mode.

Also see "Property Store" in *IBM Tivoli Directory Integrator 6.1: Users Guide* for more information about the TDI Property Store concept.

Configuration

The Server Notifications Connector uses the following parameters:

Collection
Specifies the properties file to read/write when collection type if File/URL. This parameter is required if the collection type is File/URL.

Encryption
Set to True if parser should decrypt/encrypt the entire data stream. Default value is False.

Cipher
The cipher algorithm to use when either Encryption=TRUE or the stream contains individually encrypted values. Specify SERVER to use TDI server encryption. Default cipher provided in global.properties / solution.properties

Password
The secret key to use when encrypting/decrypting the stream/property values.

Required when encryption is active and Cipher is not SERVER.

AutoRewrite
If true, the Connector will write back the contents if any auto-encrypted values were found.

System Queue Connector

Introduction

The System Queue provides a Java Message Service (JMS) like subsystem for IBM Tivoli Directory Integrator. It is designed for storing and forwarding general messages and TDI Entry Objects, between TDI 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, refer to the System Queue section in the *IBM Tivoli Directory Integrator 6.1: Administrator Guide*.

The System Queue Connector can be used with AssemblyLines in Iterator and AddOnly modes:

- In Iterator mode, the Connector retrieves TDI Entry objects from a specified message queue.
- In AddOnly mode, the Connector stores TDI 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 TDI 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. This 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 TDI server and a System

Queue Connector deployed on machine A, which work through the remote Server API with the TDI server and System Queue on machine B, which interface with the JMS server deployed on machine C.

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 TDI server or with the System Queue of a remote TDI 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 TDI server.

  The value `remote` specifies that the Connector will use the remote Server API interfaces and will work with the System Queue of a remote TDI server. In that case, the RMI URL parameter is required.

RMI URL

This is the Remote Method Invocation (RMI) URL used to connect to the remote TDI Server system. An example value 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

The `username` parameter is used to authenticate to the remote TDI 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

The password parameter is used to authenticate to the remote TDI server. This parameter is taken into account only if the `connectionType` parameter is set to `remote`.

Queue Name

This parameter 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

This parameter 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.

Detailed Log

This parameter turns on debug messages This parameter is globally defined for all TDI components

Security and Authentication

Encryption

When the connection type is set to `remote` and the remote TDI server is configured to use Secure Sockets Layer (SSL), then the System Queue Connector uses SSL, provided that a trust store on the local TDI Server is configured properly. When SSL is used, the Connector uses a Server API SSL session, which runs RMI over SSL.

Authentication

User name and password authentication: The System Queue Connector can use the remote Server API user name and password authentication. The Connector does not implement any authentication itself. The user name 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 TDI 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 TDI 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 TDI Server.

MQe Initialization

System Queue Connector and MQe Password Store Connector There are two Connectors that can initialize MQ Everyplace (MQe): “MQe Password Store Connector” on page 137 and the “System Queue Connector” on page 146. Because there can be only one instance of MQe in an instance of the JVM, it is necessary that MQe be initialized only once. To resolve the conflict that might arise by the MQe Password Store Connector and the System Queue Connector initializing MQe separately, the initialization of MQe takes place central location: the System Queue Engine.

Note: Because of these limitations, you should not attempt to start MQe yourself from within the same JVM that TDI runs in (for example from Java or JavaScript), otherwise problems will occur if you attempt to use any of the aforementioned Connectors.

Components that need MQe will be able to get the necessary MQe objects from the SystemQueueEngine.

MQe is initialized on TDI server startup. To ensure compatibility with the SystemQueue Connector, the MQePasswordStore Connector has been modified so that it retrieves the required MQe objects from the System Queue instead of initializing MQe itself.

If you need to change the default MQe parameters, you would use the MQe Configuration Utility. This utility is described in a section named "MQe Configuration Utility" in the *IBM Tivoli Directory Integrator 6.1: Administrator Guide*.

Windows Users and Groups Connector

The Windows Users and Groups Connector (in older versions of TDI 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 153 for a full functional specification of the Connector, architecture description as well as hardware and software requirements.

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 (member of the local Administrators group) and have log in 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 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 the Advanced Link Criteria (see ″Advanced link criteria″ in *IBM Tivoli Directory Integrator 6.1: 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 *ComputerName* machine with this user name and the password specified by the **Password** parameter.

Password
The value of this parameter is taken in 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 field is checked, an additional log message is 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/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

Applies only for domain users (users on the Primary Domain Controller machine). A user must always be a member of his Primary Group. This means that the value set to the **PrimaryGroup** attribute must be present in the **GlobalGroups** attribute. However, the **PrimaryGroup** attribute can be set with no value when adding a user, then default Primary Group is set to the user (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

Go 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.

Functionality

Extract user/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 the IBM Tivoli Directory Integrator engine.

Add user/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/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/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 CloudScape or DB2 8.1, Oracle or MS SQL*Server 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 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 (CloudScape syntax):

```
SQL1 = "CREATE TABLE {0} (ID VARCHAR(512) NOT NULL, ENTRY BLOB )";
SQL2 = "ALTER TABLE {0} ADD CONSTRAINT {0}_PRIMARY Primary Key (ID)";
```

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 TDI `global.properties` file and set it to *true* , then TDI 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 TDI 6.1, 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*.)

Note:
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, CPR tables and Property store tables.

The Connector supports both simple and advanced Link Criteria.

The System Store Connector along with the System Store provides an alternative to the Btree Connector. It is recommended to use the System Store Connector and System Store rather than the Btree Connector to store Entry Objects.

This Connector, like the JDBC Connector it is based upon, 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 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 TDI 6.0 and above, have an ″IDI_DS_″ prefix added to the *identifier* specified in ″Delta Store″ field of the Delta configuration tab. Similarly, CPR table names are prefixed with ″IDI_CP_″ and the property store table names are prefixed with ″IDI_PS_″.

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 and CPR tables.

Select Database Driver

Select the database to connect to. You can select CloudScape, DB2 or other.

Create Table Statement

The ″CREATE TABLE″ SQL statement used to create the tables in the selected data source. This depends on the choice of the driver in the ″Select Database Driver″ parameter. If the database driver specified is CloudScape or DB2 then the appropriate CREATE TABLE statements for these databases are displayed. This field is not editable if either CloudScape or DB2 is selected as database drivers. If the database driver selected is ″Other″ then you are *required* to enter the correct CREATE TABLE statement corresponding to the database that you choose to connect to.

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. This parameter 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

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 could be located on a CloudScape Database server, a DB2 Database server, an Oracle database server or an MS*SQL Server. Furthermore, if the System Store uses CloudScape, it can be configured to run in either embedded or networked mode. The only change that needs to be made while using the System Store Connector with the above mentioned configurations of System Store is the **Database** field.

The correct way to specify the database for different configurations of System Store is given below.

Note: The examples are specific to Windows.

Using System Store Connector with embedded CloudScape server configured as System Store **Database**: f::\Program Files\IBM\IBMDirectoryIntegrator/CloudScape

Note: In the embedded mode of operation, the CloudScape 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 CloudScape server configured as System Store **Database**: jdbc:derby://localhost:1527/E:\TDI\TDISysStore;create=true

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 CloudScape is configured to run in networked mode.
2. In networked mode of operation, the CloudScape server has to be started manually. For details regarding the ways in which a CloudScape server can be started in networked mode, please refer to the chapter on *System Store* in *IBM Tivoli Directory Integrator 6.1: Administrator Guide*

Using System Store Connector with DB2 8.1 server as System Store
Database: jdbc:db2://machine-name:50000/testDB

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

"Btree Object DB Connector" on page 20,
The chapter on *System Store* in *IBM Tivoli Directory Integrator 6.1: Administrator Guide*, *Appendix A — Using CloudScape Database* in *IBM Tivoli Directory Integrator 6.1: Users Guide*

RDBMS Changelog Connector

The RDBMS Changelog Connector enables IBM Tivoli Directory Integrator to detect when changes have occurred in specific RDBMS tables. Currently, setup scenarios are provided for tables in either Oracle or DB2 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 Changelog Connector, the RDBMS Changelog 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 Changelog Connector keeps track of a sequence number so that it only reports changes since the last iteration through the change table.

The RDBMS Changelog Connector uses JDBC to connect to a specific RDBMS table. See the "JDBC Connector" on page 92 for more information about JDBC driver issues.

The RDBMS Changelog Connector only operates in Iterator mode.

This connector supports Delta Tagging at the Entry level only.

The RDBMS Changelog Connector reads specific fields to determine new changes in the change table (see " "Change table format" on page 160"). The RDBMS Changelog Connector reads the next change table record, or discovers the first change table record. If the RDBMS Changelog Connector finds no data in the change table, the RDBMS Changelog Connector checks whether it has exceeded the maximum wait time. If the RDBMS Changelog Connector has exceeded the maximum wait time, the RDBMS Changelog Connector returns **null** to signal end of the iteration. If the RDBMS Changelog Connector finds no data in the change table, and has not exceeded the maximum wait time, the RDBMS Changelog Connector 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 Changelog 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).

During Checkpoint of the RDBMS Changelog Connector in Iterator mode, the change number will be saved as Connector Restart Info.

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. It is called JDBC URL in the IBM Tivoli Directory Integrator Config Editor.

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.

Schema
: The schema (that is, the owner) of the table of the database that you want to use. 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 operate on.

Remove Processed Rows
: Select to remove all previously processed table rows before the next poll attempt.

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.

State Key Persistence
: This 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
This 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
This updates the System Store when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual
This 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 Changelog Connector's *saveStateKey()* method, somewhere in your AssemblyLine.

Sleep Interval
Specify the time (in seconds) that IBM Tivoli Directory Integrator waits between polls of the change table.

Timeout
Specify 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 field is checked, an additional log message is 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 changelog 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 changelog 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 changelog connector. This will enable multiple RDBMS changelog 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

```
-- create source email table in Oracle.
---This will be the table that the RDBMS changelog 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 ( 80 ),
EMAIL ( 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,
   NULL,
  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,
  NULL,
  LPAD(TO_CHAR(ORAID.SGENERATOR002.NEXTVAL),20,'0'),
  'D',
  SYSDATE);END;
```

Creating Change table and triggers in MS SQL

```
-- Source table msid.email.
-- This will be the table that the RDBMS changelog connector will detect changes on.
CREATE TABLE msid.email
(
 NAME   VARCHAR (80),
 EMAIL   VARCHAR (80)
 )#

-- CCD table to capture changes. The RDBMS changelog connector uses the CCD table to capture all the ch
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
  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 msid.ccdemail
 (
  NAME,
  EMAIL,
```

```
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER
)
VALUES
(
 @NAME,
 @EMAIL,
 @@DBTS,
 @@DBTS,
'U',
GETDATE()
)
END
 DEALLOCATE insertedrows
END ELSE INSERT INTO msid.ccdemail(
NAME,
EMAIL,
IBMSNAP_COMMITSEQ,
IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER
)
SELECT
 I.NAME,
 I.EMAIL,
 @@DBTS,
 @@DBTS,
'U',
GETDATE() FROM inserted I
END;#
```

Creating change table and triggers in Informix

```
-- Create Source table infxid.email. This will be the table that the RDBMS changelog connector will det
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 an additional table (ibmsnap_seqtable) for processing commitseq and intentseq.
CREATE TABLE "infxid"."ibmsnap_seqtable"
(SEQ INTEGER NOT NULL,
```

```
  HEXREP CHAR(128)
)#

-- insert values into ibmsnap_seqtable. This example below shows how to insert a hex value
INSERT INTO ASN.IBMSNAP_SEQTABLE VALUES (1000000000,
X'0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2F303132333

-- procedure to capture INSERTs into email table
CREATE PROCEDURE infxid."email_ins_proc"
(
 NNAME  VARCHAR(80),
 NEMAIL  VARCHAR(80))
 DEFINE VARHEX CHAR(128);
 DEFINE I1 INTEGER;
 DEFINE I2 INTEGER;
 DEFINE X1 INTEGER;
 DEFINE X2 INTEGER;
 DEFINE X3 INTEGER;
 DEFINE X4 INTEGER;
 DEFINE Y1 INTEGER;
 DEFINE Y2 INTEGER;
 DEFINE Y3 INTEGER;
 DEFINE Y4 INTEGER;
 DEFINE NEWSYNCH CHAR(10);
 SELECT HEXREP
  INTO VARHEX
 FROM "infxid"."ibmsnap_seqtable";
 LET X1 = I1 / 268435456;
 LET X2 = I1 / 2097152;
 LET X3 = I1 / 16384;
 LET X4 = I1 / 128;
 LET Y1 = I2 / 268435456;
 LET Y2 = I2 / 2097152;
 LET Y3 = I2 / 16384;
 LET Y4 = I2 / 128;
 LET NEWSYNCH  =
 SUBSTR(VARHEX , MOD( X1, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( X2, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( X3, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( X4, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( I1, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y1, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y2, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y3, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y4, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( I2, 128 ) + 1 , 1 );
 INSERT INTO "infxid"."ccdemail"
(NAME,
 EMAIL,
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER )
 VALUES
(NNAME,
```

```
 NEMAIL,
 NEWSYNCH,
 NEWSYNCH,
 '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.VARCHAR
) ); #

-- create procedure to capture DELETEs on email table
CREATE PROCEDURE "infxid"."email_del_proc"
(
 ONAME  VARCHAR(80),
 OEMAIL  VARCHAR(80)
 )
 DEFINE VARHEX CHAR(128);
 DEFINE I1 INTEGER;
 DEFINE I2 INTEGER;
 DEFINE X1 INTEGER;
 DEFINE X2 INTEGER;
 DEFINE X3 INTEGER;
 DEFINE X4 INTEGER;
 DEFINE Y1 INTEGER;
 DEFINE Y2 INTEGER;
 DEFINE Y3 INTEGER;
 DEFINE Y4 INTEGER;
 DEFINE NEWSYNCH CHAR(10);
 SELECT HEXREP
  INTO VARHEX
 FROM "infxid"."ibmsnap_seqtable";
  LET X1 = I1 / 268435456;
 LET X2 = I1 / 2097152;
 LET X3 = I1 / 16384;
 LET X4 = I1 / 128;
 LET Y1 = I2 / 268435456;
 LET Y2 = I2 / 2097152;
 LET Y3 = I2 / 16384;
 LET Y4 = I2 / 128;
 LET NEWSYNCH  =
 SUBSTR(VARHEX , MOD( X1, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( X2, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( X3, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( X4, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( I1, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y1, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y2, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y3, 128 ) + 1 , 1 ) ||
```

```
 SUBSTR(VARHEX , MOD( Y4, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( I2, 128 ) + 1 , 1 );
 INSERT INTO "infxid"."ccdemail"
(NAME,
 EMAIL,
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER )
 VALUES
(ONAME,
 OEMAIL,
 NEWSYNCH,
 NEWSYNCH,
 '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),
)
 DEFINE VARHEX CHAR(128);
 DEFINE I1 INTEGER;
 DEFINE I2 INTEGER;
 DEFINE X1 INTEGER;
 DEFINE X2 INTEGER;
 DEFINE X3 INTEGER;
 DEFINE X4 INTEGER;
 DEFINE Y1 INTEGER;
 DEFINE Y2 INTEGER;
 DEFINE Y3 INTEGER;
 DEFINE Y4 INTEGER;
 DEFINE NEWSYNCH CHAR(10);
 SELECT HEXREP
  INTO VARHEX
 FROM "infxid"."ibmsnap_seqtable";
 LET X1 = I1 / 268435456;
 LET X2 = I1 / 2097152;
 LET X3 = I1 / 16384;
 LET X4 = I1 / 128;
 LET Y1 = I2 / 268435456;
 LET Y2 = I2 / 2097152;
```

```
 LET Y3 = I2 / 16384;
 LET Y4 = I2 / 128;
 LET NEWSYNCH  =
 SUBSTR(VARHEX , MOD( X1, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( X2, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( X3, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( X4, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( I1, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y1, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y2, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y3, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( Y4, 128 ) + 1 , 1 ) ||
 SUBSTR(VARHEX , MOD( I2, 128 ) + 1 , 1 );
 INSERT INTO "infxid"."ccdemail"
(NAME,
 EMAIL,
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER)
 VALUES
(NNAME,
 NEMAIL,
 '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 trigger for SYBASE

```
-- Create Source table sybid.email.
-- This will be the table that the RDBMS changelog 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_SYBTMSTMP 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)
)#
```

```
-- 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_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER )
 SELECT
 NAME,
 EMAIL,
@@DBTS,
 @@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_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER
)
 SELECT
 NAME,
 EMAIL,
 @@DBTS,
 @@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_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER
)
VALUES
(
 @NAME,
 @EMAIL,
 @@DBTS,
 @@DBTS,
'U',
GETDATE()
)
END
 DEALLOCATE insertedrows
END ELSE INSERT INTO sybid.CCDEMAIL(
 NAME,
 EMAIL,
 IBMSNAP_COMMITSEQ,
 IBMSNAP_INTENTSEQ,
 IBMSNAP_OPERATION,
 IBMSNAP_LOGMARKER
)
SELECT
 I.NAME,
 I.EMAIL,
 @@DBTS,
 @@DBTS,
'U',
GETDATE() FROM inserted I
END;#
```

runtime-provided Connector

A runtime provided Connector is a Connector Interface that is provided at runtime. When an AssemblyLine is started by an EventHandler or from a script, you can supply only one Connector to the AssemblyLine as a parameter. The Connector is used for those AssemblyLine Connectors configured as type **runtime provided**. You can use the supplied Connector several times in the AssemblyLine.

The following is an example of how to use a runtime provided Connector from an EventHandler:

```
var myConnector = system.getConnector ("ibmdi.FileSystem");
myConnector.setParam ("filePath", "mypath.txt");
myConnector.initialize ( null );

// Start the AssemblyLine
var al = main.startAL ( "AssemblyLine1", myConnector );
```

The following is an example that includes an initial working entry:

```
var myConnector = system.getConnector ("ibmdi.FileSystem");
myConnector.setParam ("filePath", "mypath.txt");
myConnector.initialize ( null );

var entry = system.newEntry();
entry.setAttribute ("cn", "My Name");
entry.setAttribute ("mail", "(my.name@dot.com)");

var al = main.startAL ( "AssemblyLine2", myConnector, entry );
```

Configuration

The Connector might need parameters, but the names and values of these parameters depend on the actual Connector.

See also

"IBM Tivoli Directory Integrator concepts – The AssemblyLine" in *IBM Tivoli Directory Integrator 6.1: Users Guide*.

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. Remember to copy over code from your hooks.

Predefined script objects

The result object

setStatus (code)

- **0** - End of Input
- **1** - Status OK
- **2** - Error

setMessage (text)

Error message.

The config object

This variable 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 357 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 359 for more details.

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.

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
This function is used to discover schema for a connection. 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 could 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.

According to the various modes, these are the minimum required functions you need to implement:

Table 5.

Mode	Function you must implement
Iterator	selectEntries() getNextEntry()
AddOnly	putEntry()
Lookup	findEntry()

Table 5. (continued)

Mode	Function you must implement
Delete	findEntry() deleteEntry()
Update	findEntry() putEntry() modEntry()
CallReply	queryReply()

Configuration

The Connector needs the following parameters:

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 field is checked, an additional log message is generated.

Script This tab is where you can create your own script code.

Examples

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

See also

"Script Parser" on page 247,
"Scripted FC" on page 273
"JavaScript Connector" in *IBM Tivoli Directory Integrator 6.1: 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 over a period of 13 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 6.1: 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. Unused in Trap mode.

SNMP Port
Client port (for client mode) unused 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 field is checked, an additional log message is 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.

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 EventHandler can be run in single or multithreaded mode,

depending on the value of the “network.mode” parameter. The Connector on the other hand works only in multithreaded mode as the Connector in Server Mode framework commands.

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 to 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 of course 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 into 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
This parameter 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
This parameter 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 more Log messages.

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("inp");
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 *serverMode=true* then the connection is closed between each call to the *getnext* method. If *serverMode=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).

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 Port
The TCP port number to connect or listen to (depends on the value of **servermode**).

TCP Host
The remote host to which connections are made (**servermode = false**).

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.

Server Mode
If **true**, then Iterating listens for incoming requests. If **false**, then Iterating connects to a remote server.

Use SSL
If checked, the Connector will deploy the Secure Socket Layer (SSL) on the connection.

Need Client Authentication over SSL
If checked and if SSL is enabled in Server Mode (i.e., listening for incoming connections), client authentication is necessary.

Detailed Log
If this field is checked, an additional log message is generated.

You can select a Parser for this Connector from the Parser pane, where you select a parser by clicking the bottom-right **Inherit From:** button.

See also

"File system Connector" on page 73,
"Direct TCP /URL scripting" on page 23,
"TCP Server Connector"
"URL Connector" on page 182.

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 which can be matched to the configured TDI trust store.

Detailed Log
If this field is checked, an additional log message is 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 deprecated 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 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 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 178

Timer Connector

The timer waits for a specified time; then it returns from sleep and resumes an AssemblyLine, i.e. 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.

Previous versions of this Connector used a Unix crontab-style **schedule** parameter to set up the exact time at which to run the Connector; when such a schedule is encountered in a Config file it is automatically converted to the new format as outlined below.

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)

Hour The hour at which to run the Timer Connector (* = any)

Minute
The minute at which to run the Timer Connector

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

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 field is checked, an additional log message is 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 73,
"TCP Connector" on page 178,
"Direct TCP /URL scripting" on page 23.

Web Service Receiver Server Connector

The Web Service Receiver Server Connector is part of the TDI Web Services suite.

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 TDI 6.1 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 TDI 6.1 configurations if there is an operation named "Default", the corresponding operation in the WSDL file is named as the name of the AssemblyLine.

- In TDI 6.1 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 TDI 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"* Attrribute, and thus to the web service client in case a request for the WSDL was received.

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 (i.e., "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 TDI 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 TDI installation folder.

Require Client Authentication
This parameter specifies whether this Connector will require clients to authenticate with client SSL certificates. If the value of this parameter is **true** (i.e., 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:

- 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 (i.e., the value is **true**, no other Attributes will be set).

When reaching the Response channel 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 15.

---

z/OS Changelog Connector

The z/OS 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.

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. .

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
: The authentication method. Possible values are:
  - CRAM-MD5 (use the CRAM-MD5 (RFC-2195) SASL mechanism).
  - none (use no authentication (**anonymous**)).
  - simple (use weak authentication (cleartext password)).
  - SASL

  If not specified, default (simple) is used. If **Login username** and **Login password** are blank, then **anonymous** is used.

Use SSL
If Use SSL is **true** (i.e., 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
This parameter 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 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.

State Key Persistence
This 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
This 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
This updates the System Store with the change log number when all Connectors and other components in the AssemblyLine have been evaluated and executed.

Manual
This 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 Changelog Connector's *saveStateKey()* method, somewhere in your AssemblyLine.

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.

See also

"LDAP Connector" on page 118,
"Active Directory Changelog (v.2) Connector" on page 7,
"Exchange Changelog Connector" on page 65
"IBM Directory Server Changelog Connector" on page 87,
"Netscape/iPlanet Changelog Connector" on page 139.

Chapter 3. EventHandlers

EventHandlers are used to extend the functionality of AssemblyLines and Connectors by providing a framework to control how they are run. This framework is particularly useful when an incoming event (for example, an incoming http, an LDAP change trigger or a JMS message) can trigger the start of a number of different AssemblyLines depending on the content in the incoming data. A few EventHandlers have been pre-programmed to make things easier.

Notes:

1. EventHandlers and the EventHandler concept are deprecated since version 6.0 of IBM Tivoli Directory Integrator, and will be phased out completely in a future version of TDI
2. EventHandlers can now only be added to your solution by using **Object->New EH** in the CE.
3. You can always override EventHandler behavior by using your own scripts. (Note however. that the concept of EventHandlers will be removed in the future, so if you heavily customize one you may find that your EventHandler has disappeared when opening the Config file in a future version of TDI 6.1.)

Migration from ChangeLog EventHandlers to ChangeLog Connectors

When opening Config Files created with TDI 6.0, existing Connector references will not need migration – old configuration files will continue to work with the updated Connectors.

As the EventHandler concept is deprecated and will be removed from future versions of TDI, you might however want to migrate from deprecated Changelog EventHandlers to Changelog Connectors.

For each EventHandler a corresponding AssemblyLine must be created. Then an Iterator Connector corresponding to the EventHandler must be inserted into the AssemblyLine "Feeds" section. Then the Connector parameters must be set – this is specific for each EventHandler/Connector pair, but generally the Connector parameters must be set the same values as the corresponding EventHandler parameters (which usually have the same names).

Any processing performed in the EventHandler Action Map must be re-implemented in the AssemblyLine "Flow" section.

The functionality of the "enabled" EventHandler parameter (otherwise known as "Auto-start service") is also available for AssemblyLines. If you want your AssemblyLine to be started right after the TDI Server is started, go to the Config/AutoStart folder in the Config Editor and add your AssemblyLine.

© Copyright IBM Corp. 2002, 2006

No migration is necessary for the JNDI Connector.

EventHandler types

The following EventHandler types are included in IBM Tivoli Directory Integrator:

Standard EventHandler
The Standard EventHandler is the most used EventHandler. You can specify conditions and actions using a number of predefined conditions and actions. It also provides hooks where you can start script code for full control. See "LDAP EventHandler" on page 211 for more information.

Primitive EventHandler (simple EventHandler, trigger or port listener)
The Primitive EventHandler enables you to script everything. With this EventHandler you have full control of the EventHandler's actions but you must code the EventHandler manually. See "Generic thread (primitive EventHandler)" on page 221 and "Timer EventHandler (primitive EventHandler)" on page 221 for more information.

Advanced EventHandler
The Advanced EventHandler wraps up even more than the Standard EventHandler in the Config Editor.

When are they started?

When TDI server starts, it scans through the table of EventHandlers and checks each one for the auto startup flag. If the auto startup flag is set, the EventHandler is spawned as a thread inside the TDI server process. When the EventHandler thread stops, TDI server does not restart the EventHandler.

Note: You can force an EventHandler to start regardless of its auto start flag using command line options.

See "Starting the EventHandler" in *IBM Tivoli Directory Integrator 6.1: Users Guide* for further discussion on how to start EventHandlers from the Config Editor.

What do they do?

EventHandlers perform a variety of functions but typically they enable external events to trigger actions in the TDI server. These actions can be specific to each site and each person, but one common action can be specific to an AssemblyLine. The timer is such an example where the external event is the clock reaching a specific time. Other events are asynchronous, such as when the TCP port EventHandler waits for incoming TCP connections, or can initiate some connection and poll for events, such as the Mailbox Connector EventHandler. Consult each EventHandler's configuration for more information.

Data flow

The EventHandlers, like AssemblyLines, have a Prolog and an Epilog. The Prolog is started before the action flow of each event.

Passing input/output file names to an AssemblyLine

To pass a filename from the EventHandler to the AssemblyLine, use an **entry** object. Here the filename is called **myFileName,** and a property is used instead of an attribute:

```
var entry = system.newEntry();
entry.setProperty("inputFileName","myFileName");
// start AssemblyLine
var al = main.startAL ("myAssemblyLine", entry);
al.join (); // wait for al to finish
```

On the AssemblyLine side you have code in your Prolog, in the **Before Connectors Initialized** hook (because you want your parameters to be used when the Connectors are initialized):

```
workEntry = task.getWork(); // gets the initial entry
var FileName = workEntry.getProperty("inputFileName");
// Set the relevant parameter of the (Connector)
myFileConnector.connector.setParam("filePath",FileName);
// If you don't want the AssemblyLine to run with the intitial entry, clear it
task.setWork(null);
```

There are a couple of finer points here:

1. Clearing the **work** entry ensures that control is passed to the first Iterator. Do not clear any entries that need to be processed. If there is a valid **work** entry when the AssemblyLine is initiated, then any Iterators in the AssemblyLine are bypassed for that work cycle, and processing starts at the first non-Iterator. See ″Connector Modes″ in *IBM Tivoli Directory Integrator 6.1: Users Guide* for more information.
2. Using properties instead of attributes ensures that the AssemblyLine does not map the attribute later (automatically mapping all attributes).

EventHandler availability

The following EventHandlers are included in TDI:

- “Active Directory Changelog EventHandler” on page 194
- “Connector EventHandler” on page 198
- “DSMLv2 EventHandler” on page 199
- “Exchange Changelog EventHandler” on page 202
- “HTTP EventHandler” on page 206
- “IBM Directory Server EventHandler” on page 208
- “LDAP EventHandler” on page 211
- “LDAP Server EventHandler” on page 214
- “Mailbox EventHandler” on page 216

- "SNMP EventHandler" on page 217
- "TCP Port EventHandler" on page 220
- "Generic thread (primitive EventHandler)" on page 221
- "Timer EventHandler (primitive EventHandler)" on page 221
- "zOS LDAP Changelog EventHandler" on page 223

Migration of Changelog EventHandlers

The EventHandler concept in IBM Tivoli Directory Integrator 6.1 is deprecated, and all ChangeLog EventHandlers in particular may be removed in a future version of IBM Tivoli Directory Integrator.

Therefore, you might consider to migrate from deprecated Changelog EventHandlers to Changelog Connectors. For each EventHandler a corresponding AssemblyLine must be created. Then an Iterator Connector corresponding to the EventHandler must be inserted into the AssemblyLine "Feeds" section. Then the Connector parameters must be set – this is specific for each EventHandler/Connector pair, but generally the Connector parameters must be set the same values as the corresponding EventHandler parameters (which usually have the same names).

Any processing performed in the EventHandler Action Map must be re-implemented in the AssemblyLine "Flow" section. The functionality of the "enabled" EventHandler parameter (otherwise known as "Auto-start service") is also available for AssemblyLines. If you want your AssemblyLine to be started right after the TDI Server is started, go to the Config/AutoStart folder in the Config Editor and add your AssemblyLine.

For a more detailed description of how this is done for the Active Directory Changelog EventHandler, see "Migration from Active Directory Changelog EventHandler to Active Directory Changelog (v.2) Connector" on page 13.

Active Directory Changelog EventHandler

The Active Directory Changelog EventHandler detects and sends notification of changes that occur in Active Directory. It reports changed Active Directory objects so that other data sources can be synchronized with Active Directory. The LDAP protocol is used both for registering for change notification and retrieving changed objects.

The EventHandler uses internally the Active Directory Changelog Connector to get changed objects from Active Directory. Changed objects retrieval is based on the **uSNChanged** mechanism.

See "Active Directory Changelog (v.2) Connector" on page 7 for details about the order of changes retrieval, the structure of the delivered Entries, and specific details about handling deleted and moved objects.

The EventHandler uses the LDAPv3 Server Notification request control to block until new changes occur in Active Directory.

Behavior

When the EventHandler starts, it connects to Active Directory and retrieves all recent directory changes that have happened while the EventHandler was offline. Then it blocks, waiting for a new change in Active Directory - when this happens it retrieves all new changes, blocks again waiting for further changes, and so on. There is no risk of losing notifications when the EventHandler is not running, because each time it starts it retrieves the changes that it missed while offline.

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

If the Active Directory goes offline, the EventHandler does not stop and tries to reconnect until it either succeeds or a stop is requested.

Access to the USN synchronization values in the User Property Store

The state of synchronization at any time is represented by four update sequence number (USN) numbers:

- START_USN
- END_USN
- CURRENT_USN_CREATED
- CURRENT_USN_CHANGED

These values are packed and stored in the User Property Store. Do not change these values manually. You might want to archive the numbers corresponding to a certain stage of synchronization and later use these numbers to replay synchronization from that stage.

The following script code can be used in IBM Tivoli Directory Integrator to get USN values stored in the User Property Store:

```
// Retrieve USN values from User Property Store
var usn = system.getPersistentObject("ad_sync");
var startUsn = usn.getString("START_USN");
var endUsn = usn.getString("END_USN");
var currentUsnCreated = usn.getString("CURRENT_USN_CREATED");
var currentUsnChanged = usn.getString("CURRENT_USN_CHANGED");

main.logmsg("START_USN: " + startUsn);
main.logmsg("END_USN: " + endUsn);
main.logmsg("CURRENT_USN_CREATED: " + currentUsnCreated);
main.logmsg("CURRENT_USN_CHANGED: " + currentUsnChanged);
```

"ad_sync" is the name of a parameter already stored in the User Property Store. The EventHandler parameter **Persistent Parameter Name** specifies this value. The previous example dumps the values to the screen, however, you might want to perform other actions, such as saving values in a file and backing up this file.

The next example of script code shows how the USN values can be stored in the User Property Store:

```
// Store USN values in the User Property Store
var usn = system.newEntry();
usn.setAttribute("START_USN", startUsn);
usn.setAttribute("END_USN", endUsn);
usn.setAttribute("CURRENT_USN_CREATED", currentUsnCreated);
usn.setAttribute("CURRENT_USN_CHANGED", currentUsnChanged);
system.setPersistentObject("ad_sync", usn);
```

This code assumes that the variables *startUsn*, *endUsn*, *currentUsnCreated* and *currentUsnChanged* contain the USN numbers as strings. This example saves the USN values under the "ad_sync" parameter, so "ad_sync" must be specified in the EventHandler parameter **Persistent Parameter Name** to continue synchronization from the desired point.

Access to the runtime EventHandler's USN synchronization values

The Active Directory Changelog EventHandler provides the following public methods to access its current USN values:

public Entry getUsnValues ();

Returns an Entry object with the following attributes:

- START_USN
- END_USN
- CURRENT_USN_CREATED
- CURRENT_USN_CHANGED

The value of each of the attributes is of type **java.lang.Integer** and represents the corresponding EventHandler's USN value.

public void setUsnValues (Entry usnEntry);

Sets the EventHandler's current USN synchronization values to the values specified in the usnEntry parameter. The structure of the usnEntry parameter must be the same as the structure of the Entry returned by getUsnValues(). The values of the usnEntry attributes must be either **java.lang.Integer** or the string representations of the corresponding numbers.

Note: Be careful when changing the USN values at runtime. Specifying inconsistent values can result in improper synchronization.

Configuration

The EventHandler needs the following parameters:

LDAP URL
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
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
The credentials (password).

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

Authentication Method
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.

LDAP Search Base
The Active Directory sub-tree that is polled for changes. For example, **dc=your_domain,dc=com**.

Note: This must be a Naming Context in the directory.

Persistent Parameter Name
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 either **EOD** or **0**. **EOD** means report only changes that occur after the EventHandler 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 **Persistent Parameter Name** parameter is not found in the User Property Store.

Detailed Log
Specifies whether more detailed debug information is written to the log file.

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

See also

"Migration of Changelog EventHandlers" on page 194.

Connector EventHandler

This EventHandler uses any Connector as an input event generator. The EventHandler calls the Connector's getNext method to obtain the next entry from the Connector. When a Connector reaches end of input, it returns null. This EventHandler continues to call the Connector's input method even after the Connector returns null. For some Connectors this can make sense whereas for others it does not.

For example, using the File System Connector makes sense because the file read by the Connector can have data appended to it at any time. Connectors selecting a finite set of entries eventually run out of entries and the EventHandler continues forever waiting for new data.

Configuration

This EventHandler needs the following parameters:

Global Connector
The Connector to use for input.

Poll Interval
The number of seconds between each call to the Connector after a **NULL** entry has been received from the Connector.

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

Detailed Log
Specifies whether more detailed log information is written to the log file.

Objects/properties/attributes

The EventHandler sets the following event properties:

event.originator
The EventHandler object

event.connector
The Connector object used by this EventHandler

The event object also contains any attribute returned by the Connector.

See also

"Starting the EventHandler" in *IBM Tivoli Directory Integrator 6.1: Administrator Guide*.

DSMLv2 EventHandler

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 an EventHandler which acts as a DSML server and listens to the DSMLv2 request over HTTP (optionally taking advantage of secure communications by means of SSL). After it receives the request, it parses the request and sends the request to an AssemblyLine to process. The result is sent back to the client over HTTP.

The DSMLv2 EventHandler is a simple Web server that provides a way to process a DSMLv2 request that is transferred as HTTP body in an HTTP request message. The DSMLv2 EventHandler automatically parses the client request into an event object and calls a user-defined AssemblyLine configured for the operation of the request. This AssemblyLine often consists of one single Connector in the appropriate mode. See "Configuration" on page 200. Also, the EventHandler implements HTTP basic authentication if an authenticator Connector is specified.

In this release, the DSML implementation does not comply fully with the DSMLv2 Specification. It supports limited bindings and operations, and is expressly designed for usage with ITIM only.

An example DSMLv2 EventHandler can be found in the examples directory (<*TDI_ installation directory*>\examples\event_handler_dsmlv2_http\DSML_EH_Test.xml).

Transportation (binding)

DSMLv2 requests and responses are transported over HTTP as HTTP body to or from the DSML Client and the DSMLv2 EventHandler. SOAP and File bindings are not part of this EventHandler.

EventHandler Workflow

Here is how the EventHandler processes a DSMLv2 request:

1. The DSMLv2 EventHandler receives a batch DSML request as a single event.
2. Each individual request from the batch request is parsed by the DSMLv2 Parser into a single Entry and fed to the AssemblyLine configured to process the operation type of that individual request (search, modify, add, etc.).
3. Result Entry(ies) generated by an AssemblyLine is (are) parsed into individual DSMLv2 response message which is accumulated.
4. When all individual requests are processed and the resulting DSML response messages are created, a batch response message containing all individual response messages is returned by the EventHandler to the client.

For RootDSE search requests, the EventHandler never initiates an AssemblyLine. It returns a list of Root Naming Contexts that are configured.

The DSMLv2 EventHandler ignores and does **not** execute the ″Action Map″ workflow.

Operations

The DSMLv2 EventHandler uses internally the DSMLv2 Parser to parse and create DSML messages. Therefore it supports the following DSML operations: **Modify**, **Add**, **Delete**, **Search**, **ModifyDN** and **Compare**.

Configuration

The EventHandler needs the following parameters:

HTTP Port
The TCP port on which this Event Handler will be listening.

Auth Realm
The authentication realm sent to the client when requesting authentication.

Auth Connector
The authenticator Connector. If you specify a Connector, it must exist in your Connector library and be configured for Lookup.

Note: Do not use any of the possible Hooks in the Auth Connector, as the Connector is called internally by the EventHandler and is not executed in the context of an AssemblyLine; the normal environment for Hooks does not apply.

This EventHandler issues authentication requests to any client (for example, Web browser) that tries to access this service. When the client provides the username and password, the EventHandler calls the authenticator Connector's Lookup method providing the username and password attributes. Therefore, your authenticator Connector must be configured using a Link Criteria where you use the $username and $password. A typical link criteria might be:

```
username equals $username
password equals $password
```

If the search fails, the EventHandler denies the request and sends an authentication request back to the client. If the search succeeds, the authentication is considered successful and your code in the EventHandler is processed. You can access the username by retrieving the HTTP attribute or property **http.remote_user**. You can access the entry returned by the authenticator Connector by retrieving the event Property **auth.entry**, by using code similar to the following:

```
var auth = event.getProperty("auth.entry");
var fullName = auth.getString("FullName");
```

Headers As Properties
If this checkbox is checked, all HTTP headers are accessible using the getProperty method of the event object. If not checked, all HTTP headers appear as attributes (for example, **getAttribute**).

Use SSL
Check this checkbox if you want to use SSL.

Note: In order to use SSL, you must generate your own certificate in your keystore (with keytool). The client must import this certificate.

Chunked Transfer Coding
If this field is checked, the body of the response message is transferred as a series of chunks.

Note: The Chunked Transfer Coding is incompatible with ITIM.

Binary Attribute
This field is used to specify user-defined binary attributes. Each attribute is checked to determine if it is a binary attribute. If it is a binary attribute, it is decoded before sending it to the AssemblyLine and encoded (if it is not encoded already) before sending the response to the client.

Naming Context
The root naming context to be associated with the set of operation AssemblyLines (specified below). Value for the Distinguished Name (dn) in the request is used to match the proper Root Naming Context. More than one naming context can be specified and each one has its own set of AssemblyLines associated.

AssemblyLine for search
The name of the AssemblyLine to be used for a search operation for the selected Root Naming Context. All Entries returned from the AssemblyLine's iterations are accumulated and returned to the client as multiple DSML search result entries.

AssemblyLine for add
The name of the AssemblyLine to be used for an add operation for the selected Root Naming Context.

AssemblyLine for modify
The name of the AssemblyLine to be used for a modify operation for the selected Root Naming Context. If Update LDAP Connector is used to modify an entry in LDAP server, Link criteria must be:

```
$dn equals $$dn
```

AssemblyLine for delete
The name of the AssemblyLine to be used for delete operation for the selected Root Naming Context. The link criteria of a Delete LDAP Connector in the AssemblyLine must be:

```
$dn equals $$dn
```

AssemblyLine for compare
The name of the AssemblyLine to be used for a compare operation for the selected Root Naming Context. The AssemblyLine does compare the value of the matching entry. If Lookup LDAP Connector is used to compare, you can use the code similar to the following in the **In Prolog->Before Initialize** hook:

```
name = work.getString("dsml.compare_name") + "";
value = work.getString("dsml.compare_value") + "";
```

and in the **Data Flow->Overwrite Lookup** hook:

```
if (compare_conn.connector.compare($dn,name,value))
 conn.setAttribute ("dsml.compare_result", "true");
else
 conn.setAttribute ("dsml.compare_result", "false");
```

AssemblyLine for modify DN
The name of the AssemblyLine to be used for a modifyDN operation for the selected Root Naming Context. If Update LDAP Connector is used to modify the DN of an entry in LDAP server, Link criteria must be:

```
$dn equals $$dn
```

In the Output Map, you add $dn to be modified.

Auto-start Service
If this checkbox is checked, this EventHandler is started when the IBM Tivoli Directory Integrator Server instance is started.

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

Comment
A comment for your own use.

Exchange Changelog EventHandler

The Exchange Changelog EventHandler detects changes that occur in Exchange Directory Service and notifies a user about these changes. It reports changed Exchange objects so that other data sources can be synchronized with Exchange.

Note: The "Exchange Changelog EventHandler"is deprecated for this release, and will be removed in future versions of TDI.

The LDAP protocol is used for retrieving changed objects.

The EventHandler uses the Exchange Changelog Connector internally to get changed objects from Exchange Directory Service. Changed objects retrieval is based on the ″USN-Changed″ mechanism.

See "Exchange Changelog Connector" on page 65 for details about the order of changes retrieval, structure of the delivered Entries, and specific details about handling deleted objects.

Notes:

1. Exchange Changelog EventHandler works with Exchange 5.5 only. If you are attempting to connect to Exchange 2000, use the Active Directory Changelog EventHandler instead.
2. The Exchange 5.5 Service Pak 4 must be installed on the Exchange Server.

Behavior

When the EventHandler starts, it connects to Exchange Directory Service and retrieves all recent directory changes that have happened while the EventHandler was offline. Then the EventHandler sleeps for a configurable period of time, then it again polls Exchange for new changes, and so on.

Notifications are not lost when the EventHandler is not running. Each time the EventHandler is started, it retrieves the changes that it missed while it was offline.

The Exchange Changelog EventHandler can be interrupted at any time during the synchronization process. The EventHandler 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 that the EventHandler is started, the EventHandler successfully continues the synchronization from the point when it was interrupted.

In case Exchange goes offline the EventHandler does not stop and tries to reconnect until either it succeeds or a stop is requested.

Access to the USN synchronization values in the User Property Store

The state of synchronization at any time is represented by four USN numbers:

- START_USN
- END_USN
- CURRENT_USN_CREATED
- CURRENT_USN_CHANGED

These values are packed and stored in the User Property Store. Do not change these values manually. You might want to archive the numbers corresponding to a certain stage of synchronization and later use these numbers to replay synchronization from that stage.

The following script code can be used in IBM Tivoli Directory Integrator to get USN values stored in the User Property Store:

```
// Retrieve USN values from User Property Store
var usn = system.getPersistentObject("exchange_sync");
var startUsn = usn.getString("START_USN");
var endUsn = usn.getString("END_USN");
var currentUsnCreated = usn.getString("CURRENT_USN_CREATED");
var currentUsnChanged = usn.getString("CURRENT_USN_CHANGED");

main.logmsg("START_USN: " + startUsn);
main.logmsg("END_USN: " + endUsn);
main.logmsg("CURRENT_USN_CREATED: " + currentUsnCreated);
main.logmsg("CURRENT_USN_CHANGED: " + currentUsnChanged);
```

"exchange_sync" is the name of a parameter already stored in the User Property Store. The EventHandler parameter **Persistent Parameter Name** specifies this value. The previous example dumps the values to the screen, however, you might want to perform other actions such as saving values in a file and backing up this file.

The next example of script code shows how the USN values can be stored in the User Property Store:

```
// Store USN values in the User Property Store
var usn = system.newEntry();
usn.setAttribute("START_USN", startUsn);
usn.setAttribute("END_USN", endUsn);
usn.setAttribute("CURRENT_USN_CREATED", currentUsnCreated);
usn.setAttribute("CURRENT_USN_CHANGED", currentUsnChanged);
system.setPersistentObject("exchange_sync", usn);
```

This code assumes that the variables *startUsn*, *endUsn*, *currentUsnCreated* and *currentUsnChanged* contain the USN numbers as strings. This example saves the USN values under the `"exchange_sync"` parameter, and so `"exchange_sync"` must be specified in the EventHandler's **Persistent Parameter Name** parameter to continue synchronization from the desired point.

Access to the runtime EventHandler's USN synchronization values

The Exchange Changelog EventHandler provides the following public methods to access its current USN values:

public Entry getUsnValues ();

Returns an Entry object with the following attributes:

- START_USN
- END_USN
- CURRENT_USN_CREATED
- CURRENT_USN_CHANGED

The value of each of the attributes is of type **java.lang.Integer** and represents the corresponding EventHandler's USN value.

public void setUsnValues (Entry usnEntry);

Sets the EventHandler's current USN synchronization values to the values specified in the usnEntry parameter. The structure of the usnEntry parameter should be the same as the structure of the Entry returned by getUsnValues(). The values of the usnEntry attributes must be either **java.lang.Integer** or the string representations of the corresponding numbers.

Note: Be careful when changing the USN values at runtime. Specifying inconsistent values can result in improper synchronization.

Configuration

The EventHandler needs the following parameters:

LDAP URL

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
The distinguished name used for authentication to the service. For example, **cn=administrator,ou=domain_name,o=organization_name**.

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

Login password
The credentials (password).

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

Authentication Method
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 Exchange Server.

LDAP Search Base
The specified Exchange sub-tree which is polled for changes. For example, **cn=recipients,ou=domain_name,o=organization_name**.

Persistent Parameter Name
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 either **EOD** or **0**. **EOD** means report only changes that occur after the EventHandler is started. **0** means perform full synchronization, that is, report all objects available in Exchange Directory Service. This parameter is taken into account only when the parameter specified by the **Persistent Parameter Name** parameter is not found in the User Property Store.

"**Is-Deleted**" **Attribute visible**
Specifies whether the Exchange server exposes the **Is-Deleted** object attribute through LDAP.

Note: If the server does expose the **Is-Deleted** attribute, but "**Is-Deleted**" **Attribute visible** is set to **false**, then the EventHandler still works properly, but you can accelerate the EventHandler by setting "**Is-Deleted**" **Attribute visible** to **true**. If the server does not expose the **Is-Deleted** attribute, but "**Is-Deleted**" **Attribute**

visible is set to **true**, then the EventHandler cannot distinguish between a modified object and a deleted object and reports all deletions as modify operations.

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

Detailed Log
Specifies whether detailed debug information is written to the log file.

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

See also

"Migration of Changelog EventHandlers" on page 194.

HTTP EventHandler

The HTTP EventHandler is a simple Web server that provides a simpler way to deal with HTTP connections than the TCP EventHandler. The HTTP EventHandler automatically parses the client request into an **event** object using the HTTP Parser. In addition the EventHandler implements HTTP basic authentication if you specify an authenticator Connector.

When dealing with events, the EventHandler forks new instances of itself, so the Prolog and Epilog are started once for each HTTP event received.

To provide simple Web server functionality, you need to provide the **http.body** and **http.content-type** attributes that the EventHandler returns as the response to a request. You can also add HTTP headers by setting any *http.** attribute. For example, setting the value for **http.my-header** causes this EventHandler to generate a **my-header: value** in the response.

The **http.body** property can be set to any of the following

Any string (java.lang.String)
The string is sent ″as is″ in the request.

Any Input stream (java.io.InputStream)
The input stream is buffered into memory to compute the *content-length* HTTP header. The input stream data is sent ″as is″ in the request.

A Java file object (java.io.File)
The *content-length* is generated by getting the file size from the file object. Then the contents of the file is sent ″as is″ in the request.

Example

The following example returns any file the client requests:

```
var base = event.getProperty("http.base");
if ( base == "/" )
 base = "/index.html";

// Construct the full path
path = "/home/httpd/documents" + base;

// Construct the Java file object
file = new java.io.File ( path );

// Set the response property
event.setProperty ( "http.body", file );
```

Configuration

HTTP Port

The TCP port on which this handler is listening.

Auth Connector

The authenticator Connector. If you specify a Connector it must exist in your Connector library and be configured for Lookup. This EventHandler issues authentication requests to any client (for example, Web browser) that tries to access this service. When the client provides the username or password, the EventHandler calls the authenticator Connector's Lookup method providing the username and password attributes. Therefore, your authenticator Connector must be configured using a Link Criteria where you use the $username and $password. A typical link criteria might be:

```
username equals $username
password equals $password
```

If the search fails, the EventHandler denies the request and sends an authentication request back to the client. If the search succeeds, the request is granted and your code in the EventHandler is executed. You can access the username by retrieving the HTTP attribute or property **http.remote_user**. You can access the entry returned by the authenticator Connector by retrieving the Property **auth.entry**, using code similar to the following:

```
var auth = event.getProperty("auth.entry");
var fullName = auth.getString("FullName");
```

Headers As Properties

If checked, all HTTP headers are accessible using the getProperty method of the event object. If not, all HTTP headers appear as attributes (for example, **getAttribute**).

Use SSL

Check the checkbox if you want to use SSL.

Note: In order to use SSL, you must generate your own certificate in your keystore (with keytool). The client must then import this certificate.

Comment

A comment for your own use.

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

Detailed Log
If this field is checked, an additional log message is generated.

See also

"EventHandler" in *IBM Tivoli Directory Integrator 6.1: Users Guide*,
"HTTP Parser" on page 242.

IBM Directory Server EventHandler

The IBM Directory Server EventHandler uses LDAP unsolicited event notifications to detect changes in an LDAP directory. To use the IBM Directory Server EventHandler your LDAP server must support LDAPv3 unsolicited notification events.

When the EventHandler starts, it connects to the LDAP server and retrieves all recent directory changes, which have happened while the EventHandler was offline, and registers for receiving unsolicited event notifications. When an event occurs in the LDAP directory, the EventHandler receives an unsolicited notification event and retrieves the next changelog entry. This changelog entry is accessible as the **event** entry object. The **event** entry object has the following attributes:

changenumber
The change number as assigned by the supplier. This integer must increase as new entries are added, and always be unique within a given server.

This attribute is Required.

targetdn
The distinguished name (DN) of the entry which was added, modified, or deleted; in the case of a modrdn operation, the **targetdn** gives the DN of the entry before it was modified.

This attribute is Required.

changetype
The type of change (**add**, **delete**, **modify**, or **modrdn**).

This attribute is Required.

changes
The changes that were made to the directory server. These changes are in LDIF format; available when **changetype** is either **add or modify**.

This attribute is Optional.

newrdn
The new Relative Distinguished Name (RDN) of the entry. If the **changeType** is

modrdn, or if the **changeType** attribute does not have the **modrdn** value, then there are no values contained in the **newrdn** attribute.

This attribute is Optional.

deleteoldrdn
A flag which tells whether the old RDN of the entry must either be retained as a distinguished attribute of the entry or be deleted.

This attribute is Optional.

newsuperior
If present, it gives the name of the entry which becomes the immediate superior of the existing entry.

This attribute is Optional.

changetime
The time when the change was made.

This attribute is Required.

modifiersname
The DN making the change.

This attribute is Optional.

An important feature of the IBM Directory Server EventHandler is that notifications are not lost when the EventHandler is not running or waiting for an action to complete, because each time it is invoked it retrieves the changes that it has missed while being offline, and iterate through them, simulating the reception of an event for every change.

Note: Even though the EventHandler will not miss any notifications as outlined above, some thought needs to be given as to how much work is performed to process each event. Starting an AssemblyLine to process a single change may not be a viable strategy in certain high volume scenarios. In such a case, it may be better to base your solution upon using an LDAP ChangeLog Connector.

Configuration

LDAP URL
The LDAP URL (ldap://*hostname*:*port*)

Login username
The distinguished name used for authentication to the server (for example, **cn=root**).

Note: This distinguished name must have administrator privileges because the EventHandler must be able to read the changelog.

Login password
The credentials (password).

ChangeLog Search Base
The search base where the changelog is stored. The standard DN for this is **cn=changelog**.

Search Base
The base of the directory tree branch about which you want to be notified. 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 Scope
The scope of events which you want to be notified about. Can be one of subtree, level and base.

ChangeNumber Filename
The name of the file where the last changenumber is stored. The file format is readable text. This file is updated after each event notification.

InitialChangeNumber
If the file supplied in the **ChangeNumber Filename** parameter does not exist, the EventHandler retrieves the changelog entries, starting from **InitialChangeNumber**.

Comment
A comment for your own use.

Authentication Method
The authentication method. Possible values are:

- MD5-CRAM (use CRAM-MD5 (RFC-2195))
- SASL (use SASL)
- Anonymous (use no authentication)
- Simple (use weak authentication (cleartext password))

If not specified, the default (**Anonymous**) is used. If either the **Login username** or **Login password** parameter is blank, then **Anonymous** is used.

Use SSL
Specifies whether to use SSL for communication with the LDAP server.

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

Detailed Log
If this field is checked, an additional log message is generated.

See also

"EventHandler" in *IBM Tivoli Directory Integrator 6.1: Administrator Guide*,
"LDAP EventHandler" on page 211,
"Migration of Changelog EventHandlers" on page 194.

LDAP EventHandler

This EventHandler uses the LDAP event notification mechanism to detect changes in an LDAP directory. To use this EventHandler, your LDAP server must support Persistent Search. The only LDAP server tested with this EventHandler is the Netscape/iPlanet/SunONE directory server (see the 119 for more about iPlanet), but other LDAP servers may work as well.

Note: If you can base your change detection in your LDAP database upon reading a Changelog (using an appropriate Changelog Connector) instead of relying upon the events signalled by the LDAP database by means of the Persistent Search mechanism, your solution is much more robust. The Persistent Search mechanism is very ephemeral: you only see changes if you are actually listening while the changes occur. If the connection is down you miss them. A Changelog, on the other hand, is backed by mass storage, and given the right System Change Number you can always pick up processing from where you last stopped.

When the EventHandler starts, it connects to the LDAP server and specifies the selection criteria for event notifications. All DNs returned from the EventHandler are relative to the search base specified. To construct the full DN in a flexible way, you can append the search base to, for example, the new DN with the following code in a custom script

```
event.setProperty("ldap.newdn", event.getProperty("ldap.newdn") +
  "," + task.getParam("ldapSearchBase"));
```

When an event occurs in the LDAP directory, the EventHandler sets the **ldap.operation** property to one of the following values:

objAdded
: A new entry was added to the directory.

objRenamed
: An existing entry was renamed.

objModified
: An existing entry's attributes were modified.

objRemoved
: An existing entry was removed.

handleError
: An error was encountered.

Depending on the **ldap.operation**, the EventHandler sets the following properties:

Object Added (_objAdded)

ldap.newdn
: The new DN in case of a rename operation

ldap.newentry
The new entry with changes applied

Object Rename (_objRenamed)

ldap.dn
The old DN

ldap.newdn
The new DN

Object Modified (_objModified)

ldap.dn
The DN before the modify operation.

ldap.entry
The contents of the LDAP entry before the modify operation. This functionality is only available for LDAP databases where a modification operation is done by first removing the object and then recreating it with the modified attributes.

ldap.newdn
The DN after the modify operation.

ldap.newentry
The contents of the LDAP entry after the modify operation.

Object Removed (_objRemoved)

ldap.dn
The DN before the remove operation

ldap.entry
The contents of the LDAP entry before the remove operation

The **ldap.entry** and **ldap.newentry** properties are instances of the Entry class so you can access these as you normally do with **conn** and **work** objects in the AssemblyLine as shown in the following example:

```
var old = event.getProperty ("ldap.entry");
task.logmsg ("Old common name = " + old.getString("cn") );
```

Note: One important aspect of the LDAP EventHandler is that you can lose important notifications when the EventHandler is not running. This EventHandler is best used when you want to trap changes in a directory but still can tolerate loss of information.

Error Encountered (_handleError)

ldap.error
The java exception thrown by the EventHandler.

Note: iPlanet Directory 5.0 has changed the changelog to a proprietary format. Go to the following URL:

```
http://docs.iplanet.com/docs/manuals/
 directory/51/html/ag/replicat_new.htm#1)
```

You must install the Retro ChangeLog Plug-in for accessing the change log. The following is an extract from the Change Log section of the iPlanet documentation:

"In iPlanet Directory Server 5.0, the format of the change log was modified. In earlier versions of Directory Server, the change log was accessible over LDAP. Now, however, it is intended only for internal use by the server. If you have applications that need to read the change log, you need to use the Retro Change Log Plug-in for backward compatibility. For more information, refer to the Retro Change Log Plug-In."

Configuration

The EventHandler needs the following parameters:

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 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
This parameter is only used if the Connector is in Iterator mode. The possible values are:

subtree
Return entries on all levels from search base and below.

onelevel
Return entries that are immediately below searchbase only.

Comment
A comment for your own use.

Authentication Method
Type of LDAP authentication. Can be one of the following:

- Simple (using ldapUsername/ldapPassword). Treated as anonymous if username or password are not provided)
- MD5-CRAM

- SASL
- Anonymous (treated as Simple if username and password are supplied)

Use SSL
If this is checked, use secure sockets layer for communication with the LDAP server.

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, use this flag, as the value of an attribute cannot be an empty string.

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

Detailed Log
If this field is checked, an additional log message is generated.

See also

"EventHandler" in *IBM Tivoli Directory Integrator 6.1: Users Guide*,
"IBM Directory Server Changelog Connector" on page 87.

LDAP Server EventHandler

The LDAP Server EventHandler accepts an LDAP connection request from an LDAP client. The LDAP Server EventHandler generates a copy of itself to take care of this connection until the connection is closed by the LDAP client. The LDAP Server EventHandler only terminates when the TCP connection is closed. Each LDAP message received on the connection drives one cycle of the LDAP Server EventHandler logic. The main thread returns to listening for similar LDAP requests from other LDAP clients. The LDAP operation in the message is parsed into the LDAP Server EventHandler entry object. The LDAP Server EventHandler code is executed, and the return message is built 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 EventHandler 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 EventHandler **work** entry. 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. To facilitate scripting, the parser copies the LDAP message into the LDAP Server EventHandler **work** object.

Scripting

The LDAP Server EventHandler must do work to determine the desired outcome of the LDAP message. The code can reside in the LDAP Server EventHandler, or the LDAP Server EventHandler can start an AssemblyLine to do the work. The basic LDAP operations

(**SEARCH**, **BIND**, **UNBIND**, **COMPARE**, **ADD**, **DELETE**, **MODIFY**, and **MODIFYRDN**) are provided as values in the LDAP Server EventHandler 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 EventHandler. 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 LDAP Server EventHandler responds to each request by setting the **ldap.status**, **ldap.matcheddn** and **ldap.errormessage** entry attributes. **ldap.matcheddn** and **ldap.errormessage** are optional.

At the end of the LDAP Server EventHandler execution cycle, the LDAP Server EventHandler 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 EventHandler terminates the connection and records an error if the received message does not conform to the LDAP v3 format

Note: The LDAP Server EventHandler does not perform any validation on the incoming attributes. Any operation or parameter value is therefore accepted.

Configuration

The EventHandler needs the following parameters:

LDAP Port
The TCP port on which this EventHandler listens.

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.

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

Detailed Log
If this field is checked, an additional log message is generated.

Use SSL
If checked the server will only accept SSL connections.

Note: Depending on your solution implementation, you may need to change the port number as well.

Mailbox EventHandler

This EventHandler listens for changes in a mailbox. Depending on the protocol the handler either polls the mailbox periodically by reconnecting to the mailbox (POP3) or periodically issues idle messages on the connection (IMAP4).

Configuration

This EventHandler needs the following parameters:

Server Name
The mail server hosting the mailbox.

Protocol
Specify POP3 or IMAP.

Login Username
The user name.

Login Password
The password for **Login Username**.

Mail Folder
The mail folder to monitor. For POP3 this can only be INBOX. For IMAP4 servers this can be any folder available on the server.

Poll Interval (seconds)
Number of seconds between each poll. Be aware that for POP3 this incurs a new connection each time.

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

Detailed Log
If this field is checked, an additional log message is generated.

Objects/properties/attributes

The EventHandler sets the following event properties:

event.originator
The EventHandler object.

mailbox.session
The Java session object (javax.mail.Session).

mailbox.store
The message store object (javax.mail.Store).

mailbox.folder
The folder object (javax.mail.Folder).

mailbox.message
The message object (javax.mail.Message).

mailbox.operation
The operation related to mailbox.message. For pop3 connections only *existing* entries are reported. For imap connections this property contains the value *new* or *deleted*.

mail.subject
The subject header from the mail.message.

mail.from
The from header from the mail.message.

mail.to
The first recipient in the mail.message.

Examples

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

See also

"EventHandler" in *IBM Tivoli Directory Integrator 6.1: Users Guide*.

SNMP EventHandler

SNMP is a standard wire-level protocol used to query and set published attributes in remote systems. Typically, it is used by a monitoring console to configure and get status information from a wide range of SNMP-compliant systems.

Note: This implementation of the SNMP EventHandler supports protocols up to version 2.

The SNMP EventHandler receives UDP packages on a specified port, and returns an appropriately formatted response to the originator. The implementation is non-blocking by default, meaning the SNMP EventHandler spawns a copy of itself to perform the work while the main thread returns to listen for further packets (the main thread can be set to blocking as well).

Setting the community string enables the SNMP EventHandler to ignore incoming SNMP requests that do not contain this specific string. Leaving the community string blank results in all SNMP packets arriving in the SNMP EventHandler.

The value of the SNMP operation is provided in the **SNMP.operation** attribute in the SNMP EventHandler **work** entry. Legal values are **GET**, **GETNEXT**, **SET** and **TRAP**. A single SNMP message can request an operation on multiple OIDs (OID is an address into 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.

To facilitate scripting, the parser copies the OID table into two multi-valued attributes in the SNMP EventHandler **work** object: **SNMP.OID**, containing the desired OID, and **SNMP.OIDvalue**, which contains the corresponding value. **OIDValue** contains **java.lang.Integer** or **java.lang.String** values. After parsing, the SNMP EventHandler **work** object contains the following:

- The community-string in **SNMP.community**
- The originating IP Address in **SNMP.remoteIP**
- The SNMP sequencing number in **SNMP.requestId**
- The attributes **SNMP.errorcode** (value set to **2**) and **SNMP.errorindex** (value set to **0**)

TRAP messages contain an additional set of values:

- **SNMP.enterprise** – OID for object-generating trap
- **SNMP.agentAddress** – address of object-generating trap
- **SNMP.specificTrap**, **SNMP.genericTrap** – has value when **specificTrap** = **6**
- **SNMP.timeTicks** – time since last initialization of object-generating trap

Scripting the desired action

The SNMP EventHandler must do work to determine the desired outcome of the SNMP message. The code can reside in the SNMP EventHandler, or the SNMP EventHandler can start an AssemblyLine to get the work done.

The basic SNMP operations (**GET**, **GETNEXT**, **SET** and **TRAP**) are provided as values in the SNMP EventHandler scripting environment to facilitate scripting, for example, if **SNMP.operation** equals **GET**. By modifying the **SNMP.OID** and **SNMP.OIDvalue** attributes, you can build content that is sent back in the return message. The return error code is manipulated by setting the value of the SNMP EventHandler **SNMP.errorcode** attribute and **SNMP.errorindex** (the entry in the OID table that contains the error).

Error handling

SNMP supports return error codes that are part of the **GET-RESPOND** return message. Messages that do not conform to the SNMP format are not processed by the SNMP EventHandler, but control is given to the Epilog, where a user can add customized code:

- The default error code is **2**, indicating that no work has been done. An error code **0** indicating success must be specifically set by the user.
- Only standard SNMP-compliant error codes can be set by the user. The SNMP EventHandler overwrites an error code with the value **5** (interpreted as other error) if the value is outside the supported range before turning the message to the originator.

Returning the SNMP packet returned values

The SNMP EventHandler formats and returns some of the attributes of the **work** entry. These are:

- **SNMP.OID**
- **SNMP.OIDValue**

  Note: Only **string**, **integer** and **NULL** are supported. **java.lang.String** is mapped to **OctetString** and **java.lang.Integer** is mapped to **integer**.
- **SNMP.errorcode**

  Note: The default value is **2**. If the user tries to set **SNMP.errorcode** to something outside the range **0**-**5**, **SNMP.errorcode** is set to **5** before being returned.
- **SNMP.errorindex**

  Note: Set to the (zero-based) index in the OID table of the OID value that caused the error.

Note: According to the SNMP protocol, only one error can be reported. In case of multiple errors, **SNMP.errorindex** must be the lowest index indicating an error. The user is responsible to set this value correctly.

Configuration

This EventHandler needs the following parameters:

UDP Port
: The SNMP default UDP port for get/set operations is 161, but is configurable. TRAPs are usually received on port 162.

Verify Community
: If set, discard all messages not matching this community string. If blank, enable all community strings.

Multi threaded
: Check to create a new thread for each event.

Auto-start Service
: If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

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

TCP Port EventHandler

This EventHandler 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 EventHandler goes back to listening mode. When the newly created thread has completed, the thread stops and the TCP connection is closed.

This EventHandler is forking, so the Prolog and Epilog are started once for each HTTP event received.

Note: The TCP Port EventHandler is deprecated for release 6.1 of IBM Tivoli Directory Integrator, and will be removed in a future release. Build your solution using the TCP Server Connector instead.

Configuration

TCP Port
The TCP port on which to listen for incoming connections.

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

Detailed Log
If this field is checked, an additional log message is generated.

Objects/properties/attributes

The EventHandler sets the following event properties:

event.originator
The EventHandler object

event.inputstream
TCP socket input stream

event.outputstream
TCP socket output stream

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)

Examples

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

See also

"EventHandler" in *IBM Tivoli Directory Integrator 6.1: Users Guide*.

Generic thread (primitive EventHandler)

The generic thread is started at startup and continues to run as long as the script runs. The script can call the **task.sleep(milliseconds)** to periodically perform its work.

Configuration

The port listener needs the following parameters:

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

Detailed Log
Specifies whether more detailed log information is written to the log file.

Script The script to run.

See also

"Mailbox EventHandler" on page 216.

Timer EventHandler (primitive EventHandler)

The timer waits for a specified time, when it starts a script or starts an AssemblyLine. The script must be provided by the administrator or user.

Configuration

This EventHandler needs the following parameters:

Auto-start Service
If this field is checked, this EventHandler is started when IBM Tivoli Directory Integrator is started.

Schedule
This parameter decides when the EventHandler is run. The format is as follows:

```
<month> <day> <weekday> <hour> <minute>
```

The fields have numeric values:

- Month = 0 – 11 (January . . . December)
- Day = 1 – 31
- Weekday = 1 – 7 (Sunday . . . Saturday)
- Hour = 0 – 23
- Minute = 0 – 59

Fields are separated by white space. Enter "*" to specify any value. You can specify multiple values for any field, separated by commas, but the values must be in ascending order.

When the current time matches all the fields in the schedule, the specified AssemblyLine is run. For example:

- * * 5 22 0 - Run every Thursday at 22:00 hours
- * 3 * 22 0 - Run every 3rd of each month at 22:00 hours

Notes:

1. The month field has values from 0 to 11, while day and weekday values begin at 1.
2. A common source of confusion is specifying both a day and a weekday. Both attributes must match, meaning that this event does not occur often.

Run AssemblyLine
The AssemblyLine to start.

Detailed Log
If this field is checked, an additional log message is generated.

Script If specified, the script must contain a function called **ontimer**. This function is called with no parameters whenever the time specified by the schedule parameter is reached, and then the AssemblyLine is started. The schedule of the EventHandler can be modified through the **timer** object. You reconfigure the timer by setting the schedule parameter from the **ontimer** function. For example:

```
function ontimer()
{
timer.setParam ("schedule", "* * * 22 0");
}
```

Examples

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

zOS LDAP Changelog EventHandler

The zOS LDAP Changelog EventHandler iterates through the change records in the zOS LDAP server (RACF accessed through its LDAP interface). This EventHandler is very similar to the IBM Directory Server EventHandler (see "IBM Directory Server EventHandler" on page 208). The difference is that this EventHandler polls the changelog instead of using Unsolicited Event Notification.

Note: The zOS LDAP Changelog EventHandler does not currently support Unsolicited Event Notification.

The zOS LDAP server creates change records when certain events happen in its LDAP Directory (**add**, **modify**, **delete**, and so forth). This EventHandler is used to iterate through those change records and initiate AssemblyLines as configured by the user.

Use this EventHandler as you might use the IBM Directory Server EventHandler. The zOS LDAP Changelog EventHandler can be used just as effectively with the IBM Directory Server instead of the existing IBM Directory Server EventHandler.

Configuration

The EventHandler needs the following parameters:

LDAP URL
The URL to reach the zOS LDAP server. This URL is in the format: ldap://*hostname.domain*:*port*

Login username
The DN to authenticate to the LDAP server.

Login password
The password to authenticate to the LDAP server (for **Login username**).

ChangeLog Search Base
The suffix used for changelog. In the case of zOS, it is always **cn=changelog** (**cn=changelog** is the default).

Search Base
The part of the directory tree that you are interested in changing. For example, if search base is **o=ibm,c=us**, the EventHandler responds to all changes in the **o=ibm,c=us** tree, but not those in **o=ms,c=us**.

Search Scope
The scope to use in conjunction with search base (the default is subtree). See previous example (**Search Base**). If scope was set to **baseObject**, changes to the given object only are triggered.

ChangeNumber Filename
The file to store the last processed change number in (this is a plain text file).

InitialChangeNumber
If **ChangeNumber Filename** does not exist, this number is used as the starting point for the changelog processing. If **ChangeNumber Filename** exists, this field is ignored.

PollInterval
The time (in seconds) that the EventHandler sleeps between successive polls to the changelog.

Comment
Write any comments you want.

Authentication Method
The type of authentication method to use for the LDAP connection.

Use SSL
Initiates SSL to the LDAP server (be sure to change port in **LDAP URL**).

Auto-start Service
Check this checkbox if you want this EventHandler to start automatically when the server starts.

Detailed Log
Turns on debug logging in the EventHandler.

Polling logic

Because the zOS LDAP server does not currently support Event Notification, a polling mechanism is used to check for new entries in the changelog.

When the EventHandler starts, it attempts to read the last processed change number from the file specified in the configuration. If this file does not exist, the **InitialChangeNumber** as specified in the **Config ...** panel is used.

A root DSE search is done to retrieve the **lastchangenumber** attribute. If the current **changenumber** (from file or **Config ...**) is less than the **lastchangenumber**, then the EventHandler uses an LDAP Connector to request the next sequential **changenumber** from zOS LDAP.

If the entry described in that changelog entry fits within the search boundaries, the event is dispatched and the **changenumber** is incremented. If the entry does not fit the search boundary, or the entry does not exist, the **changenumber** is incremented and the entry is ignored.

This comparison against the **lastchangenumber** is continued until the **changenumber** is equal to the **lastchangenumber**. When this condition is true, it means that the last **changenumber** in the system and the EventHandler goes to sleep for the **PollInterval** as specified in the **Config ...** panel. After this interval has passed, the root DSE is searched again for **lastchangenumber** and the cycle is repeated.

If at any time the **lastchangenumber** attribute is equal to **0**, meaning that the changelog is empty, the EventHandler sleeps for the **PollInterval**.

See also

"Migration of Changelog EventHandlers" on page 194.

Chapter 4. Parsers

Parsers are used in conjunction with a transport Connector to interpret or generate the content that travels over the Connector's byte stream.

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

- "CSV Parser" on page 228
- "DSML Parser" on page 229
- "DSMLv2 Parser" on page 230
- "Fixed Parser" on page 242
- "HTTP Parser" on page 242
- "LDIF Parser" on page 245
- "Line Reader Parser" on page 246
- "Script Parser" on page 247
- "Simple Parser" on page 250
- "SOAP Parser" on page 250
- "XML Parser" on page 252
- "XML SAX Parser" on page 256
- "XSL based XML parser" on page 258

Character Encoding conversion

Java2 uses Unicode as its internal Character Encoding. 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 conversion. When you read from text files on the local system, Java2 has already established a default Character Encoding conversion that is dependent on the platform you are running.

The TDI 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. The Connectors that require a Parser usually accept a **characterSet** parameter in the Parser configuration. If set, this parameter must be set to one

© Copyright IBM Corp. 2002, 2006

of the accepted conversion tables found in the Java2 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 chapter.

Availability

Please refer to the IANA Charset Registry (http://www.iana.org/assignments/character-sets).

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 File 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

This parameter specifies the character used to separate each column. 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.

Field Names

This parameter 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

When this parameter is set to **true**, the field is output with quotes around it under the same conditions as in previous versions, however, quotes 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, quotes 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 quotes.

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.

Character Encoding
Character Encoding conversion. This parameter is not directly available from the Parser configuration. You must edit the configuration file manually to set it.

Detailed Log
If this field is checked, an additional log message is generated.

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 conversion.

Detailed Log
If this field is checked, an additional log message is 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

"XML Parser" on page 252,
"SOAP Parser" on page 250,
"DSMLv2 Parser."

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.

An example can be found in the examples directory (<*installation directory*>\examples\event_handler_dsmlv2_http\DSML_EH_Test.xml).

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 238.

Mode

The DSMLv2 Parser operates either in Server or in Client mode:

- In Server mode the Parser reads/parses DSMLv2 requests and write/creates DSMLv2 responses
- In Client mode the Parser reads/parses DSMLv2 responses and writes/creates DSMLv2 requests.

Operations

The DSMLv2 Parser supports **Modify**, **Add**, **Delete**, **Search**, **ModifyDN**, **Compare**, **Auth** and **Extended** operations.

Attention: The following TDI 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 these raw request and response objects in 5.1 are no available through the other Attributes delivered by the DSMLv2 Parser and DSMLv2 EventHandler.

Modify Request

Entries with the following structure are parsed (on read) and created (on write) by the parser for Modify Requests:

- attribute ″dsml.operation″ – set to ″modifyRequest″
- attribute ″dsml.base″ – holds the ″dn″ XML attribute of the DSML ″modifyRequest″ element
- attribute ″$dn″ – holds the ″dn″ XML attribute of the DSML ″modifyRequest″ element
- for each modification item: a TDI attribute named as the ″name″ XML attribute of the DSML ″modification″ element, with the values specified for the ″modification″ DSML element and TDI 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:

- attribute ″dsml.operation″ – set to ″modifyResponse″
- attribute ″$dn″ – holds the ″matchedDN″ XML attribute of the DSML ″modifyResponse″ element
- attribute ″dsml.resultcode″ – holds the ″code″ XML attribute of the ″resultCode″ XML element of the DSML response
- attribute ″dsml.resultdescr″ – holds the ″descr″ XML attribute of the ″resultCode″ XML element of the DSML response

- attribute ″dsml.error″ – the presence of this attribute indicates an error condition and holds the value of the ″errorMessage″ XML element of the DSML response
- attribute “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.
- attribute “dsml.referral” – holds 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:

- attribute ″dsml.operation″ – set to ″searchRequest″
- attribute ″$dn″ – holds the ″matchedDN″ XML attribute of the DSML ″compareResponse″ element
- attribute ″dsml.base″ – holds the ″dn″ XML attribute of the DSML ″searchRequest″ element
- attribute ″dsml.scope″ – holds the value of the ″scope″ attribute of the DSML ″searchRequest″ element
- attribute ″dsml.filter″ – the LDAP filter that corresponds to the ″filter″ element of the DSML request
- attribute ″dsml.attributes″ – the value of this attribute is a Vector whose elements hold the names of the attributes listed in the ″attributes″ element of the DSML request.
- attribute “dsml.derefAliases” – holds the value of the “derefAliases” attribute of the DSML “searchRequest” element
- attribute “dsml.sizeLimit” – holds the value of the “sizeLimit” attribute of the DSML “searchRequest” element
- attribute “dsml.timeLimit” – holds the value of the “timeLimit” attribute of the DSML “searchRequest” element
- attribute “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:

- attribute ″dsml.operation″ – set to ″searchResponse″
- attribute ″$dn″ – holds the ″matchedDN″ XML attribute of the DSML ″searchResultDone″ element of the DSML response
- attribute ″dsml.resultcode″ – holds the ″code″ XML attribute of the ″resultCode″ XML element of the DSML response
- attribute ″dsml.resultdescr″ – holds the ″descr″ XML attribute of the ″resultCode″ XML element of the DSML response

- attribute ″resultEntries″ – a multivalued attribute, each of its values is an TDI Entry whose attributes correspond to the ″attr″ elements of the corresponding ″searchResultEntry″ element.
- attribute ″dsml.error″ – the presence of this attribute indicates an error condition and holds the value of the ″errorMessage″ XML element of the DSML response
- attribute “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:

- attribute ″dsml.operation″ – set to ″addRequest″
- attribute ″dsml.base″ – holds the ″dn″ XML attribute of the DSML ″addRequest″ element
- attribute ″$dn″ – holds the ″dn″ XML attribute of the DSML ″addRequest″ element
- for each DSML attr element: a TDI 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:

- attribute ″dsml.operation″ – set to ″addResponse″
- attribute ″$dn″ – holds the ″matchedDN″ XML attribute of the DSML ″addResponse″ element
- attribute ″dsml.resultcode″ – holds the ″code″ XML attribute of the ″resultCode″ XML element of the DSML response
- attribute ″dsml.resultdescr″ – holds the ″descr″ XML attribute of the ″resultCode″ XML element of the DSML response
- attribute ″dsml.error″ – the presence of this attribute indicates an error condition and holds the value of the ″errorMessage″ XML element of the DSML response
- attribute “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.
- attribute “dsml.referral” – holds 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:

- attribute ″dsml.operation″ – set to ″deleteRequest″
- attribute ″dsml.base″ – holds the ″dn″ XML attribute of the DSML ″delRequest″ element
- attribute ″$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:

- attribute ″dsml.operation″ – set to ″deleteResponse″
- attribute ″$dn″ – holds the ″matchedDN″ XML attribute of the DSML ″delRequest″ element
- attribute ″dsml.resultcode″ – holds the ″code″ XML attribute of the ″resultCode″ XML element of the DSML response
- attribute ″dsml.resultdescr″ – holds the ″descr″ XML attribute of the ″resultCode″ XML element of the DSML response
- attribute ″dsml.error″ – the presence of this attribute indicates an error condition and holds the value of the ″errorMessage″ XML element of the DSML response
- attribute “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.
- attribute “dsml.referral” – holds 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:

- attribute ″dsml.operation″ – set to ″modDnRequest″
- attribute ″dsml.base″ – holds the ″dn″ XML attribute of the DSML ″modDNRequest″ element
- attribute ″$dn″ – holds the ″dn″ XML attribute of the DSML ″modDNRequest″ element
- attribute ″newrdn″ – holds the ″newrdn″ XML attribute of the DSML ″modDNRequest″ element
- attribute “dsml.newSuperior″ – holds the “newSuperior” XML attribute of the DSML “modDNRequest” element
- attribute “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:

- attribute ″dsml.operation″ – set to ″modDnResponse″

- attribute ″$dn″ – holds the ″matchedDN″ XML attribute of the DSML ″modDNResponse″ element
- attribute ″dsml.resultcode″ – holds the ″code″ XML attribute of the ″resultCode″ XML element of the DSML response
- attribute ″dsml.resultdescr″ – holds the ″descr″ XML attribute of the ″resultCode″ XML element of the DSML response
- attribute ″dsml.error″ – the presence of this attribute indicates an error condition and holds the value of the ″errorMessage″ XML element of the DSML response
- attribute “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.
- attribute “dsml.referral” – holds 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:

- attribute ″dsml.operation″ – set to ″compareRequest″
- attribute ″dsml.base″ – holds the ″dn″ XML attribute of the DSML ″compareRequest″ element
- attribute ″$dn″ – holds the ″dn″ XML attribute of the DSML ″compareRequest″ element
- attribute ″dsml.compare_name″ – holds the ″name″ XML attribute of the ″assertion″ element of the DSML request
- attribute ″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:

- attribute ″dsml.operation″ – set to ″compareResponse″
- attribute ″$dn″ – holds the ″matchedDN″ XML attribute of the DSML ″compareResponse″ element
- attribute ″dsml.compare_result″ – either ″true″ of ″false″ depending on whether the compare found match or not. When the Parser is used to create DSML response, this attribute is required and depending on its value the Parser sets the right result code value.
- attribute ″dsml.resultcode″ – holds the ″code″ XML attribute of the ″resultCode″ XML element of the DSML response
- attribute ″dsml.resultdescr″ – holds the ″descr″ XML attribute of the ″resultCode″ XML element of the DSML response

- attribute ″dsml.error″ – the presence of this attribute indicates an error condition and holds the value of the ″errorMessage″ XML element of the DSML response
- attribute “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.
- attribute “dsml.referral” – holds 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:

- attribute “dsml.operation” – set to ″authRequest″
- attribute ″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:

- attribute “dsml.operation” – set to ″ authResponse″
- attribute ″$dn″ – holds the “matchedDN” XML attribute of the DSML “authResponse” element
- attribute ″dsml.resultcode″ – holds the “code” XML attribute of the “resultCode” XML element of the DSML response
- attribute “dsml.resultdescr” – holds the “descr” XML attribute of the “resultCode” XML element of the DSML response
- attribute “dsml.error” – the presence of this attribute indicates an error condition and holds the value of the “errorMessage” XML element of the DSML response
- attribute “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.
- attribute “dsml.referral” – holds 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:

- attribute “dsml.operation” – set to ″extendedRequest″
- attribute “dsml.extended.requestname” – holds the “requestName” XML attribute of the DSML “extendedRequest” element

- attribute “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:

- attribute “dsml.operation” – set to ″extendedResponse″
- attribute ″$dn″ – holds the “matchedDN” XML attribute of the DSML “extendedResponse” element
- attribute ″dsml.resultcode″ – holds the “code” XML attribute of the “resultCode” XML element of the DSML response
- attribute “dsml.resultdescr” – holds the “descr” XML attribute of the “resultCode” XML element of the DSML response
- attribute “dsml.error” – the presence of this attribute indicates an error condition and holds the value of the “errorMessage” XML element of the DSML response
- attribute “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.
- attribute “dsml.referral” – holds Vector containing all referral URIs of the DSML “extendedResponse” element
- attribute “dsml.responseName” – holds the “responseName″ XML attribute of the DSML “extendedResponse” element
- attribute “dsml.response” – holds byte array containing string which represents the response from an ″extendedResponse″ operation

Binary and non-String Attributes

When parsing DSML messages, attributes tagged as binary by the **Binary Attributes** Parser parameter are Base64 decoded, i.e. 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:

- Entry Attribute ″dsml.requestID″ corresponds to the DSMLv2 ″requestID″ attribute.

- Entry Attribute ″dsml.controls″ holds an array of raw ″com.ibm.dsml2.parser.Control″ objects and corresponds to the ″control″ DSMLv2 elements.

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 a TDI 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 TDI-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 a TDI 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
This parameter specifies the XML character encoding; for example, UTF-8 or ASCII.

Mode This parameter 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

This parameter 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
This parameter 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
This parameter 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
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`.

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:

- 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
```

Trim values
Trim leading or trailing spaces for input fields.

Character Encoding
Character Encoding conversion.

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 **get 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 mode. If not set, the parser operates in server mode.

Character Encoding
Character Encoding conversion.

Detailed Log
If this field is checked, an additional log message is generated.

Attributes or properties

When constructing a page, depending on the value of **Headers As Properties**, the Parser uses these attributes or properties, where relevant, to construct the header. When reading a page, the Parser parses the header to fill in these attributes or properties where possible.

http.method
The HTTP method when sending this information in client mode (default is **GET**). This attribute or property is returned in server mode. See http://www.w3.org/Protocols/HTTP/Methods.html for more information about HTTP methods.

http.url
The URL to use. This attribute or property is mandatory in client mode.

http.content-type
The content type for the returned **http.body** (if any). If this is set to **application/x-www-form-urlencoded**, the **http.body** is also parsed for more headers. The default value when writing (when **http.body** contains something), is **text/plain**.

http.responseCode
The HTTP response code as an Integer object. Read in client mode.

http.responseMsg
The HTTP response message as a String object. Read in client mode.

http.content-encoding
The encoding of the returned **http.body** (if any)

http.content-length
The number of bytes in **http.body**. This attribute or property is returned when reading, and ignored when writing. It is recomputed by the Parser.

http.body
When reading, depending of the content-type of the data, this object is an instance of java.lang.StringBuffer, a char[] or a byte[] that contains the returned body. When the content is a StringBuffer, you can use code such as the following:

```
var body = conn.getObject ("http.body");
task.logmsg ("Returned text: " + body.toString() );
```

See ″Using binary values in scripting″ in *IBM Tivoli Directory Integrator 6.1: Users Guide* for information about handling a char[] or byte[]. When writing, **http.body** must contain either an instance of java.io.File, in which case that file is used as the body, or some text that is used as the body.

http.text-body
When reading, 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.

http.redirect
When this attribute or property has a value, and the user is writing and in server mode, redirect message pointing to the value of this attribute or property is sent.

http.status
Used when writing in server mode. The default is **OK**. Some possible values are:

- **OK** or **200 OK**- Returns a **200 OK** response.
- **FORBIDDEN** or **401 Forbidden**- Attempts to authenticate using 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.authorization
Contains the authorization read in server mode. It is probably easier to use **http.remote_user** and **http.remote_pass**.

http.remote_user
The username returned when reading in server mode, or the username to use when writing in client mode.

http.remote_pass
The password returned when reading in server mode, or the password to use when writing in client mode.

http.base
The base URL. Returned when reading in server mode.

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.

Character sets/Encoding

Character set when reading

The default character encoding when reading is **iso-8859–1**. This encoding is overridden by the characterSet 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 characterSet 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

See also

"HTTP Client Connector" on page 75,
"HTTP Server Connector" on page 81.

LDIF Parser

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 238.

Configuration

The Parser has the following parameters:

DN Attribute Name
The attribute name to use.

Version Number
Displays a version attribute in the beginning of the output if checked. This parameter is **On** by default.

Note: LDIF parser can now 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 conversion.

Detailed Log
If this field is checked, an additional log message is 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 "Example" on page 358 for an example of how not 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
This parameter specifies the name of the attribute that contains the line. Default is **line**.

Character Encoding
Character Encoding conversion.

Detailed Log
If this field is checked, an additional log message is 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. 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

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 (*root_directory*/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.");
}
```

Configuration

The Parser has the following parameters:

Script Language

Choose an available script language from the list.

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 conversion.

Detailed Log

If this field is checked, an additional log message is 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. Remember to copy over code from your hooks.

Example

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

See also

"Script Connector" on page 172,
"Scripted FC" on page 273
"JavaScript Parser" in *IBM Tivoli Directory Integrator 6.1: 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 conversion.

Detailed Log
If this field is checked, an additional log message is 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 EventHandler and 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 conversion.

Detailed Log
: If this field is checked, an additional log message is 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.

XML Parser

The XML Parser reads and writes XML documents. 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 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 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. If you read large (greater than 4MB) or write large (greater 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 6.1: Users Guide* for a solution to this. Alternatively, use the "XML SAX Parser" on page 256.
2. The Parser silently ignores empty entries.
3. When reading a CDATA attribute, no blank space is trimmed from the value. However, blank space is trimmed from attributes that are not CDATA.

4. 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.
5. 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");
   ```
6. The implementation of XML used in IBM Tivoli Directory Integrator uses the DOM model, which means it is memory based. The whole tree must be contained in memory for it to be valid. This is not a problem in regards to Connectors utilizing the XML Parser in Iterator mode, except possibly for the fact that you could run out of memory if the input file is large. During normal operation as well as restart the Parser reads the whole tree into memory but still skips to the correct entry if so instructed by the Checkpoint/Restart framework at restart.

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 conversion.

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 field is checked, an additional log message is generated.

Character Encoding in the XML Parser

The default and recommended Character Encoding to use when deploying the 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, the encoding specified in the XML header of the file is used. If no encoding is specified there, then the encoding specified in the parser config is used. If stilll no encoding is specified, then UTF-8 is used.
- 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 entries that look like the following:

```
<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:

```
<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 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/xmlparser directory of your IBM Tivoli Directory Integrator.

See also

"XML SAX Parser,"
"XSL based XML parser" on page 258,
"SOAP Parser" on page 250,
"DSML Parser" on page 229.

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.

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

GroupTag
XML Group tag name that encloses entries.

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
The character set to observe when reading; the default is UTF-8.

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.

See also

"XML Parser" on page 252,

"XSL based XML parser."

XSL based XML parser

Introduction

The XSL based XML DOM Parser enables TDI 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 TDI internal format. The XSL transforms the DOM Document generated from input XML, and produces an output DOM for the TDI 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 TDI 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 TDI 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.

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 TDI internal format.
- The output XSL file (when used in a Filesystem connector in AddOnly mode): to transform TDI 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

TDI 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 TDI internal format from entry object with attribute value pairs, can be formed.

See also

"XML Parser" on page 252

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-106.ibm.com/developerworks/XML/library/x-domjava/

XSL Transformation using Xalan and Java
http://www.perfectxml.com/vip1.asp

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 6.1: Users Guide* for a further explanation of this example.

Chapter 5. Function Components

Function Components (FC) are, besides of Connectors, EventHandlers 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 may present 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 TDI web service architecture.

The Function Components provided with the IBM Tivoli Directory Integrator 6.1 are:

- "AssemblyLine FC" on page 269
- "Axis EasyInvoke Soap WS FC" on page 291
- "Axis Java To Soap FC" on page 281
- "Axis Soap To Java FC" on page 290
- "CBE Generator Function Component" on page 274
- "Complex Types Generator FC" on page 294
- "Function Component For SAP R/3" on page 308
- "InvokeSoap WS FC" on page 286
- "Java Class Function Component" on page 271
- "Memory Queue FC" on page 279
- "Parser FC" on page 272
- "Remote Command Line FC" on page 296
- "Scripted FC" on page 273
- "SendEMail Function Component" on page 278
- "WrapSoap FC" on page 284
- "XMLToSDO FC" on page 264
- "SDOToXML FC" on page 266
- "z/OS TSO/E Command Line FC" on page 300

© Copyright IBM Corp. 2002, 2006

XMLToSDO FC

The EMF XMLToSDO Function Component converts an XML document to SDO objects connected in a tree-like structure resembling the XML structure.

For each XML element an XML Attribute Data Object is created. TDI 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 TDI 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 TDI 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 TDI 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

xsdFile
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”.

useNamespaces
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.

namespaceMap
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.

inputXMLType
Specifies the type of the input XML document. It can be a java.lang.String object or org.w3c.dom.Element object.

debug Turns on debug messages.

Migration

The EMF XMLToSDO and SDOToXML Function Components are not compatible with the TDI 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 TDI 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, i.e. the names of elements/attributes, etc. are specified in the XML Schema file specified as a Function Component parameter.

SDOToXML FC

The EMF SDOToXML Function Component converts Service Data Objects to XML. This component uses an XML Schema definition to build an Ecore model.

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 TDI 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 TDI 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 TDI 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). More details on the “namespaceMap” parameter can be found in section “Configuration”.

Configuration

xsdFile
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”.

useNamespaces
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”.

namespaceMap
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.

returnXMLType
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.

debug Turns on debug messages.

Using the FC

Migration

The EMF XMLToSDO and SDOToXML Function Components are not compatible with the TDI 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 TDI 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, i.e. the names of elements/attributes, etc. are specified in the XML Schema file specified as a Function Component parameter.

AssemblyLine FC

The AssemblyLine 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.

Configuration

AssemblyLine
A drop-down list of pre-defined AssemblyLines that could be the target of this FC.

Server The TDI 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)

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 6.1: 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 (e.g. 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).

Note that the 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 AL has a defined schema (AL Call/Return tab), then this will be used.
2. Otherwise the FC examines the Input and Output maps of all Connectors in the AL to be called in order to ″guess″ its schema.

Java Class Function Component

IBM Tivoli Directory Integrator 6.1 (TDI) allows you to use Java objects in your script code to perform specific operations not provided directly by TDI. 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 6.

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
This parameter specifies the fully qualified name of the Java class. This parameter is required.

Method
This parameter specifies the method to call in the Java class.

Parser FC

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

Content Parser
A drop-down list of pre-defined parsers that interpret or generate data stream records.

Read Operation mode of the Parser: Read from parser, Write to parser (returning result)

Returns result as String
Check to have the function return a String object instead of a `Bytearray` object

Detailed Log
When checked, generates additional log messages.

Comment
Your own comments go here.

Using the FC

This FC allows you to select a Parser and then set its mode to either input (*$PARSERFC_WM_READ*) or output (*$PARSERFC_WM_WRITE*).

In input mode, you must provide an attribute (Output Map) called *"value"* which is either a string, or a java.io.Stream object to be used as input for the Parser. The FC will return a **conn** object with the parsed attributes, which are then available for your Input Map.

In Output mode the FC takes the attributes passed in the Output Map and applies the Parser to them, 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 FC

Like Connectors, EventHandlers 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.

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 the parameter block as described by your FCs Config dialog.

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 parameter block as described by your FCs Config dialog.

getUI (fc)
This function implements/overrides the standard getUI() method, and can be used to configure the FC's operational parameters. This function is optional.

These correspond to the three main Function Interface methods. Each method is passed an initial Function Component parameter, which is the AL FC wrapper, giving your script access to settings like the Attribute Maps etc.

See also

"Script Connector" on page 172,
"Script Parser" on page 247,
"JavaScript Connector" in *IBM Tivoli Directory Integrator 6.1: Users Guide*.

CBE Generator Function Component

The CBE Generator Function Component allows you to generate Common Base Event (CBE) event objects which can be written to CBE logs or emitted to an IBM Common Event Infrastructure (CEI) server.

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.

CBE FC Configuration

The CBE Generator 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 .

Debug
Turns on debug messages. This parameter is globally defined for all TDI components.

Input and Output Map Attributes

A dotted notation for the attributes is supported in the attribute map for several TDI components.

Table 7. Input map attributes

Attribute Name	Attribute Type
Event	org.eclipse.hyades.logging.events.cbe.CommonBaseEvent
eventXml	String

Table 8. Output map attributes

Attribute Name	Attribute Type	Description
Creation Time	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

Table 8. Output map attributes (continued)

Attribute Name	Attribute Type	Description
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.***
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

Table 8. Output map attributes (continued)

Attribute Name	Attribute Type	Description
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
SCI		***This notation describes the source component identification. These are required properties for a CommonBaseEvent.***
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`

Generating a CBE Log XML

In one of the scripts (after event generation from CBE Generator), a call to the **getCBELogXML( )** API (exposed in the CBEGeneratorFC) 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);
```

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.

See also

- 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 TDI 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.

Configuration

The SendEMail Function Component uses the following parameters:

smtpServerHost
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.

smtpServerPort
This parameter specifies the port of the SMTP server that sends mails.

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 password used for SMTP authentication.

useSSL
This parameter uses Secure Sockets Layer (SSL) to communicate with the SMTP server.

from This parameter 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 this parameter is not set, the `recipients` Entry Attribute should be mapped.

subject
This parameter specifies the subject of the e-mail.

attachments
This parameter allows you to attach any files(s) you want to include with your message. 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
```

contentType
This parameter allows you to set the MIME content type of the e-mail's body. `text/plain` is the default value.

encoding
This parameter 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/1.3/docs/guide/intl/encoding.doc.html

debug Turns on debug messages. This parameter is globally defined for all TDI components

---

Memory Queue FC

Aka. MemQueue FC. The Memory buffer FC encapsulates the functionality of the TDI Memory Buffer Pipe (as present in the API) and provides a GUI configuring 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.

Note: The Memory Queue FC is deprecated in this release. It is much easier and also recommended to use the "Memory Queue Connector" on page 132 or directly use "The system object" on page 360 to create a new pipe, add data to the pipe and put data into the pipe. APIs for all these functionalities have been exposed in the System Object.

Configuration

Instance name
Name of the TDI 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.

Watermark
The Watermark parameter is either:

- With **Paging Off** (next parameter), it is the maximum queue size. When this size is reached, the **Blocking Add Parameter** determines if the read waits or fails since the queue is full.
- With **Paging On**, it 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.

Enable paging using System Store
Check this to let the Memory Buffer Queue use the System Store for paging.

Page Size
Number of entries in one page.

Database name
A JDBC URL of an external database to use.

Username
Login username to the database used.

Password
Login password to the database used.

Table name
Table to use for paging.

Blocking add operation
Check this option to block or fail (and always log that data is lost) while adding, if queue is full and no paging used.

Detailed Log
Check for additional log messages.

Using the FC

"The system object" on page 360 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:

```
MemoryBufferQ pipe = system.getFunction("ibmdi.MemoryBufferPipeFC").perform(null);
```

Using the *Entry* call:

```
var inp = system.newEntry();
inp.setAttribute ("test", "this is a sample entry");
MemoryBufferQ pipe = system.getFunction("ibmdi. MemoryBufferPipeFC").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 MemoryBufferQ object has two methods that can aid in managing the object: `purgeQueue()` and `deletePipe()`.

See also

"Memory Queue Connector" on page 132

"System Queue Connector" on page 146

Axis Java To Soap FC

The Axis Java-to-Soap Function Component (FC) is part of the TDI Web Services suite.

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: 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: 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 TDI.
- 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

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 the 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);
```

See Also

"Axis Soap To Java FC" on page 290

WrapSoap FC

The WrapSoap Function Component (FC) is part of the TDI Web Services suite.

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 (i.e., `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
This parameter 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: 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: 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 FC

Introduction

The Axis InvokeSoapWS Function Component (FC) is part of the TDI Web Services suite.

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 FC" on page 290 and "Axis Java To Soap FC" on page 281 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 (i.e. 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 289.)

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. If the WSDL URL (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*l" parameter. If the "*WSDL URL*" parameter is missing or empty, the 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 reqeust-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 FC" on page 291

Axis Soap To Java FC

The Axis Soap-to-Java Function Component (FC) is part of the TDI Web Services suite.

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: 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 this parameter 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 TDI.
- 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 283
"Axis Java To Soap FC" on page 281

Axis EasyInvoke Soap WS FC

The Axis EasyInvokeSoapWS Function Component (FC) is part of the TDI Web Services suite.

This is a ″simplified″ web service invocation component: it is a stand-alone FC with its own Config screen, but internally instantiates, configures and uses 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, i.e. you are tied to the processing and binding provided by Axis, but you gain simplicity of setup and use.

Authentication

The AxisEasyInvokeSoapWS FC uses org.apache.axis.client.Call's authentication mechanism. When username and password parameters of the FC are specified, then they are set to be used by the Call object. This information is sent to the server and if it requires authentication it takes this two parameters for username and password.

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.

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 parameter is required. 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.

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: 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: 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.

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 the org.apache.axis.client.Call's authentication mechanism. When the username and password parameters of the FC are filled, then they are set to be used by the Call object. This information is sent to the server and if it requires authentication it takes this 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 serv

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 TDI.
- 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.
- 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.

  Object[] –> AxisEasyInvokeSoapWS FC –> Object[]

  or

  Entry –> AxisEasyInvokeSoapWS FC –> Entry

See also

“InvokeSoap WS FC” on page 286

Complex Types Generator FC

The Complex Types Generator Function Component is part of the TDI Web Services suite.

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 (i.e., 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:

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.

Remote Command Line FC

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 v2.1 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 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.

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 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.

Configuration

Target Machine Hostname
The hostname (address) of the target machine.

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' and 'WIN'. This designates what protocol to use when connecting to the remote machine. See "Using the FC" on page 298 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 optional if an input attribute 'command.line' has been provided.

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.

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, i.e. 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.

Detailed Log
Enabling this will generate more 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.

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 Assembly Line containing other TDI 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 TDI components further down in the Assembly Line.

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 TDI 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.

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
```

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.

See also

"Command line Connector" on page 22,
"z/OS TSO/E Command Line FC."

---

z/OS TSO/E Command Line FC

This Function Component addresses the need TDI 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 that manages RACF users. This Connector could construct the correct RACF commands (that correspond to add, modify, etc.) and use internally the FC to execute them.

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.

TDI 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.

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. Copy the REXX script and the JCL to the dataset.

   For example, this can be done from the TSO shell or menu 6 of ISPF with the following commands:

   ```
   OGET '<TDI_root>/tso_fc/TDIEXEC' '<TDIEXEC_dataset>(TDIEXEC)'
   OGET '<TDI_root>/tso_fc/TDITP.jcl' '<TDITP.jcl_dataset>(TDITP)'
   ```
3. Make sure APPC and ASCH (Transaction Scheduler) are started.

   APPC could be started with the following system command:

   ```
   s APPC,SUB=MSTR
   ```

   ASCH could be started with the following system command:

   ```
   s ASCH,SUB=MSTR
   ```

   Note: 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
   ```
4. Customize TDITP.jcl to reflect your environment.

   To do so, follow the instructions within the TDITP.jcl JCL. Basically you need to specify the name of the dataset where the TDIEXEC REXX script resides, the APPC TP Profile dataset and the Transaction Scheduler class. The APPC configuration files will give you the necessary information for the APPC TP Profile dataset and the Transaction Scheduler class: by default the APPC and ASCH configuration files are located in the USER.PARMLIB dataset: APPCPM00 for APPC and ASCHPM00 for the Transaction Scheduler.
5. Submit the modified TDITP.jcl.

You can submit it from ISPF by typing ″sub″ in front of the name of the JCL.

See also

″z/OS environment Support″, in *IBM Tivoli Directory Integrator 6.1: Administrator Guide*.

Chapter 6. SAP R/3 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 R/3, and it describes the procedural steps that are required to achieve integration between IBM Tivoli Directory Integrator and SAP R/3.

This chapter assumes that both IBM Tivoli Directory Integratorand SAP R/3 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 R/3.

This chapter contains the following sub-sections:

- "Software Requirements"
- "Verifying the Component Suite for SAP R/3" on page 306
- "Checking the Version Numbers" on page 307
- "Uninstallation" on page 308

Software Requirements

The IBM Tivoli Directory Integrator Component Suite for SAP R/3 is supported on the operating systems platforms that are common for IBM Tivoli Directory Integrator and SAP JCo 2.1.6. 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 2.1.6. SAP JCo has other prerequisites, please refer to the SAP Note 684106 for information about other prerequisites.

Installing IBM Tivoli Directory Integrator 6.1 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

Licensed SAP R/3 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. SAP R/3 6.20 or 6.40 must also be installed and running on a node within the

© Copyright IBM Corp. 2002, 2006

network environment. TCP/IP network connectivity is required between the SAP R/3 instance and the machine hosting the IBM Tivoli Directory Integrator Component Suite for SAP R/3.

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 R/3 , the JCo can be installed and configured for IBM Tivoli Directory Integrator 6.1 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 R/3 system to which the IBM Tivoli Directory Integrator Component Suite for SAP R/3 will connect.
4. Copy `sapjco.jar` from the SAP JCo package directory to the *TDI_HOME*`/jars` folder.
5. **On Windows machines only,** copy `librfc32.dll` and `sapjcorfc.dll` to the *TDI_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 R/3, 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 R/3

To verify the IBM Tivoli Directory Integrator 6.1 Component Suite for SAP R/3:

Table 9 below describes the files and locations installed by the system installer for IBM Tivoli Directory Integrator 6.1, with regards to the Components Suite.

Table 9. Installed locations for the IBM Tivoli Directory Integrator Component Suite for SAP R/3

Filename	Description
`TDI_SAPR3Connectors_UserGuide.doc`	*TDI_HOME*`/doc`
`SapR3BorConnector.jar`	*TDI_HOME*`/jars/connectors`
`SapR3UserConnector.jar`	*TDI_HOME*`/jars/connectors`
`SapR3RfcFC.jar`	*TDI_HOME*`/jars/functions`
`index.html` (Javadoc for all SAP Components)	*TDI_HOME*`/docs/api/`

Table 9. Installed locations for the IBM Tivoli Directory Integrator Component Suite for SAP R/3 (continued)

Filename	Description
bapi_user_actgroups_assign.xsl	*TDI_HOME*/xsl
bapi_user_actgroups_delete.xsl	*TDI_HOME*/xsl
bapi_user_change.xsl	*TDI_HOME*/xsl
bapi_user_create.xsl	*TDI_HOME*/xsl
bapi_user_delete.xsl	*TDI_HOME*/xsl
bapi_user_getdetail_postcall.xsl	*TDI_HOME*/xsl
bapi_user_getdetail_precall.xsl	*TDI_HOME*/xsl
bapi_user_getlist_postcall.xsl	*TDI_HOME*/xsl
bapi_user_getlist_precall.xsl	*TDI_HOME*/xsl
bapi_user_profiles_assign.xsl	*TDI_HOME*/xsl
bapi_user_profiles_delete.xsl	*TDI_HOME*/xsl
bapi_employee_dequeue.xsl	*TDI_HOME*/xsl
bapi_employee_enqueue.xsl	*TDI_HOME*/xsl
bapi_employee_getdata_postcall.xsl	*TDI_HOME*/xsl
bapi_employee_getdata_precall.xsl	*TDI_HOME*/xsl
bapi_persdata_change.xsl	*TDI_HOME*/xsl
bapi_persdata_create.xsl	*TDI_HOME*/xsl
bapi_persdata_delete.xsl	*TDI_HOME*/xsl
bapi_persdata_getdetail_postcall.xsl	*TDI_HOME*/xsl
bapi_persdata_getdetail_precall.xsl	*TDI_HOME*/xsl
bapi_persdata_getdetailedlist_postcall.xsl	*TDI_HOME*/xsl
bapi_persdata_getdetailedlist_precall.xsl	*TDI_HOME*/xsl

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 R/3 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 R/3.
2. Run the uninstall executable located at *TDI_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 R/3 system to which the IBM Tivoli Directory Integrator Component Suite for SAP R/3 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_HOME*/jars folder.
7. **On Windows machines only,** remove the librfc32.dll and sapjcorfc.dll files from the *TDI_HOME*/libs folder.

Function Component For SAP R/3

This chapter describes the IBM Tivoli Directory Integrator Function Component for SAP R/3.

This chapter includes the following sections:

- "Function Component Introduction"
- "Configuration" on page 309
- "Using the Function Component" on page 312

Function Component Introduction

The Function Component for SAP R/3 6.20 and 6.40 uses SAP JCo 2.1.6 to invoke RFCs on the SAP R/3 System. The Function Component provides a means of calling an arbitrary RFC.

Figure 1 below illustrates the overview architecture of the RFC Function Component.

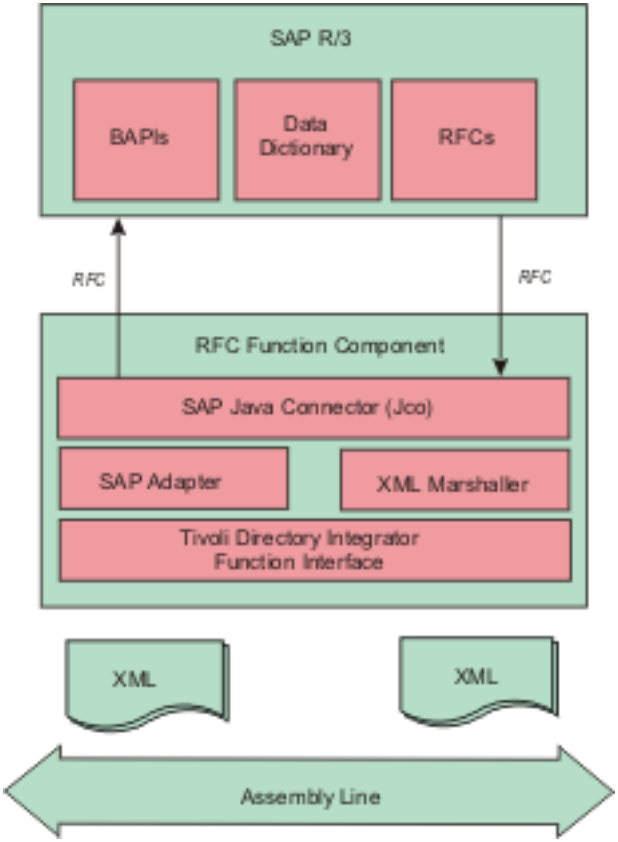


Figure 1. Overview architecture of the RFC Function Component

Before using the Function Component for SAP R/3, the SAP JCo must be downloaded and installed (for details, see "Software Requirements" on page 305).

Configuration

If the Function Component for SAP R/3 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

R3 Client (client)
SAP R/3 Logon client for R/3 connection. For example, 100.

R3 User (user)
SAP R/3 Logon user for R/3 connection.

Password (passwd)
SAP R/3 Logon password for R/3 connection.

R3 System Number (sysnr)
The SAP R/3 system number for R/3 connection. For example, 00.

R3 Hostname (ashost)
SAP R/3 application server name for R/3 connection.

Gateway host (gwhost)
Gateway host name for R/3 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 R/3. 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);
```

Note: For SAP specifications, see the SAP Web site at http://ifr.sap.com.

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 R/3 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 R/3 system.

To invoke the utility, add *TDI_HOME*/jars/functions/SapR3RfcFC.jar to the CLASSPATH environment variable:

```
TDI_HOME/_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 306. 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_HOME*/_jvm/jre/bin/java.exe

The contents of the JCO Properties file represent the R/3 client connection parameters for the R/3 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 R/3

The section describes the configuration and operation of the IBM Tivoli Directory Integrator User Registry Connector for SAP R/3.

This chapter contains the following sections:

- "Introduction"
- "Configuration" on page 315
- "Using the User Registry Connector for SAP R/3" on page 319

Introduction

This component enables the provisioning and management of SAP R/3 user accounts to external applications (with respect to SAP R/3). The Connector uses the generic RFC invocation feature of the IBM Tivoli Directory Integrator Function Component for SAP R/3 (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 R/3 client application.

The Connector supports an extendable generic framework for provisioning SAP R/3 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 R/3 instance.

The key features and benefits of the Connector are:

- Support for Create, Read, Update, and Delete (C.R.U.D) operations for SAP R/3 users.
- Modifiable behavior through XSL transformations for SAP R/3 RFC execution.
- Minimal compile time dependency between the Connector and SAP R/3. 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 R/3 User Registry.

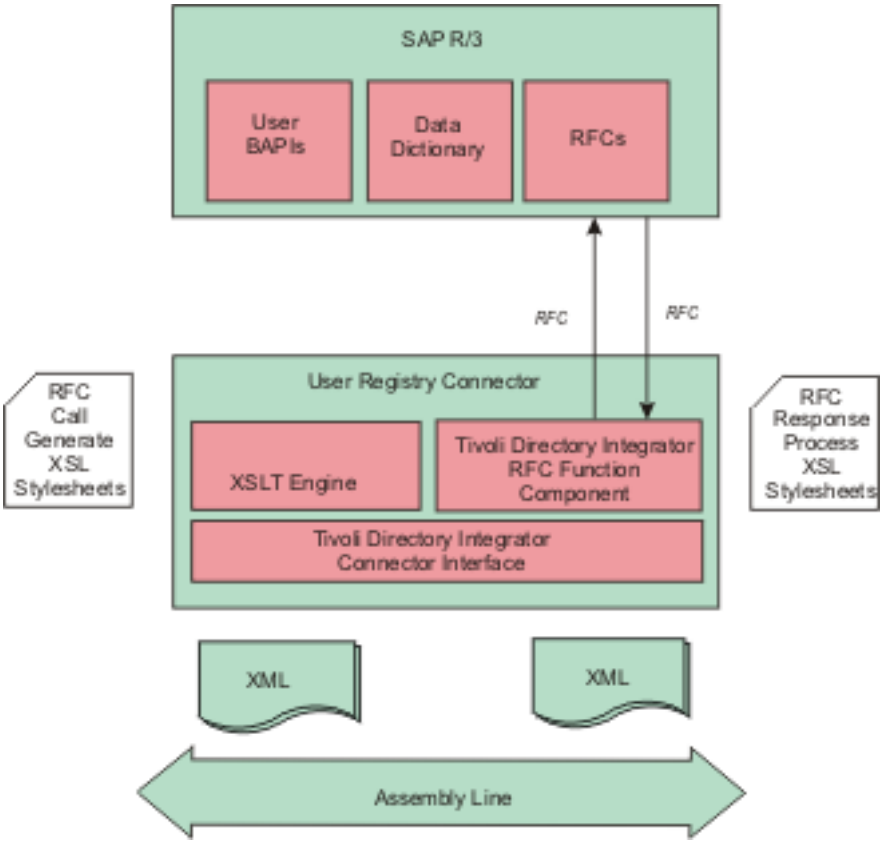


Figure 2. Component design of the SAP R/3 User Registry

Configuration

The User Registry Connector for SAP R/3 may be added directly into an assembly line. The following section lists the configuration parameters that are available for R/3 client connections and XSL style sheet behavior. The runtime names are shown in parentheses.

Parameters

R3 Client (client)

SAP R/3 Logon client for R/3 connection (for example, *100*). This is passed directly to the RFC Function Component.

R3 User (user)

SAP R/3 Logon user for R/3 connection. This is passed directly to the RFC Function Component.

Password (passwd)
SAP R/3 Logon password for R/3 connection. This is passed directly to the RFC Function Component.

R3 System Number (sysnr)
The SAP R/3 system number for R/3 connection (for example, *100*). This is passed directly to the RFC Function Component.

R3 Hostname (ashost)
SAP R/3 application server name for R/3 connection. This is passed directly to the RFC Function Component.

Gateway host (gwhost)
Gateway host name for R/3 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 R/3. 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, e.g. 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 p
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 R/3

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 Assembly Line 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 R/3 has been installed in (or if the Assembly Line 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 10. IBM Tivoli Directory Integrator Schema

Attribute Name	Type	Description
sapUserXml	java.lang.String	A string representing the attributes of an R/3 user. The XSchema is defined in "XSchema for User Registry Connector XML" on page 341. 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.
sapUserName	java.lang.String	A string representing the name of a given SAP R/3 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 R/3 database. The Connector should be added to the **Flow** section of a IBM Tivoli Directory Integrator Assembly Line. 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 R/3 database. The Connector should be added to the **Flow** section of a IBM Tivoli Directory Integrator Assembly Line. 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 Assembly Line, since the Assembly Line 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 R/3 database. The Connector should be added to the **Flow** section of a IBM Tivoli Directory Integrator Assembly Line. The **sapUserName** attribute must be defined in the **Link Criteria** of the Connector. The **Link Criteria** is required by the Assembly Line, since the Assembly Line

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 R/3 user. The Connector should be added to the **Flow** section of a IBM Tivoli Directory Integrator Assembly Line. 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 Assembly Line 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 R/3 database, in turn, and make those details available to the Assembly Line. 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 Assembly Line will first call the Connector's **selectEntries()** method to obtain and cache a list of all user names in the SAP R/3 database. 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. 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 R/3. 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 Assembly Line 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 Assembly Line, 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 Assembly Line. 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 R/3

This section describes the configuration and operation of the IBM Tivoli Directory Integrator Human Resources/Business Object Repository Connector for SAP R/3.

This chapter contains the following sections:

- "Introduction"
- "Configuration" on page 327
- "Using the Human Resources Connector for SAP R/3" on page 330

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 11. Infotype Number Ranges

Number Range	HR Submodule
0000 to 0999	HR Master Data

Table 11. Infotype Number Ranges (continued)

Number Range	HR Submodule
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.

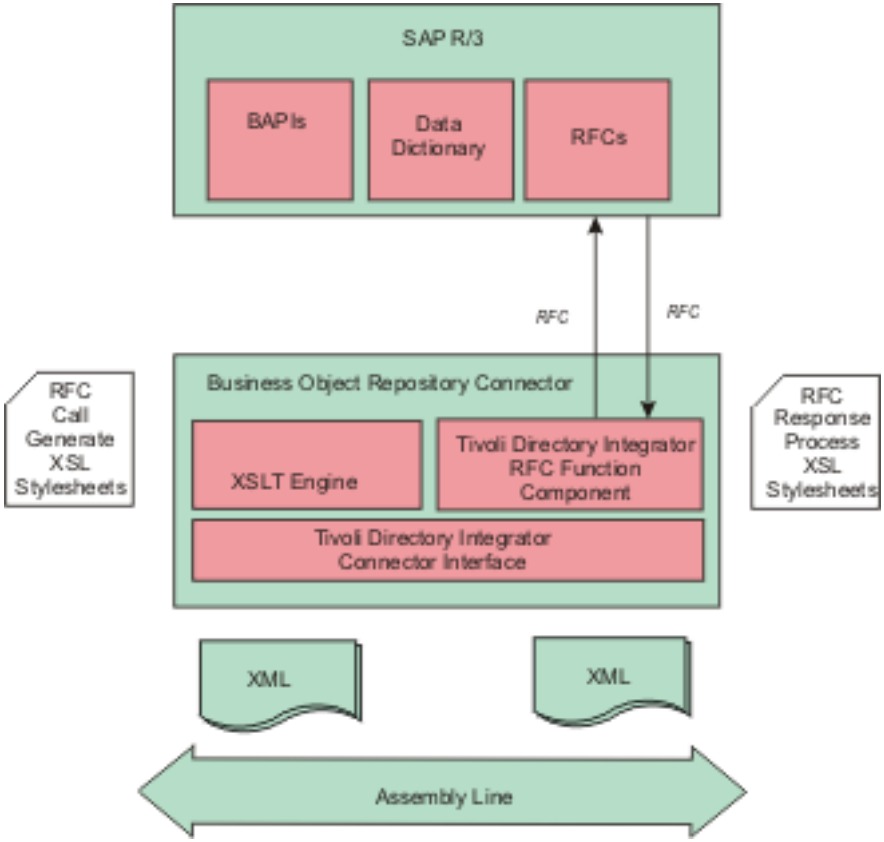


Figure 3. Component design of the Human Resources/Business Object Repository Connector for SAP R/3

The integration distribution package 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 R/3.

The key features and benefits of the Connector are:

- Support for Create, Read, Update, and Delete (C.R.U.D) operations for SAP R/3 HR data.
- Modifiable behavior through XSL transformations for SAP R/3 RFC execution.
- Minimal compile time dependency between the Connector and SAP R/3. 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 12. 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>
```

Configuration

The BOR Connector for SAP R/3 may be added directly into an assembly line. The following section lists the configuration parameters that are available for R/3 client connections and XSL style sheet behavior. Runtime names are shown in parentheses.

Parameters

R3 Client (client)
SAP R/3 Logon client for R/3 connection (for example, *100*). This is passed directly to the RFC Function Component.

R3 User (user)
SAP R/3 Logon user for R/3 connection. This is passed directly to the RFC Function Component.

Password (passwd)
SAP R/3 Logon password for R/3 connection. This is passed directly to the RFC Function Component.

R3 System Number (sysnr)
The SAP R/3 system number for R/3 connection (for example, *100*). This is passed directly to the RFC Function Component.

R3 Hostname (ashost)
SAP R/3 application server name for R/3 connection. This is passed directly to the RFC Function Component.

Gateway host (gwhost)
Gateway host name for R/3 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 R/3. 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, e.g. 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 pos
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 R/3. 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 R/3

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 Assembly Line 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 R/3 has been installed in (or if the Assembly Line 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 13. Entry Schema Attributes

Attribute Name	Type	Description
sapXml	java.lang.String	A string representing the attributes of an R/3 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 13. 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.

Table 13. Entry Schema Attributes (continued)

Attribute Name	Type	Description
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 R/3 database. The Connector should be added to the **Flow** section of a Tivoli Directory Integrator Assembly Line. 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 R/3 database. The Connector should be added to the **Flow** section of a Tivoli Directory Integrator Assembly Line. 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 Assembly Line, since the Assembly Line 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 R/3 database. The Connector should be added to the **Flow** section of a Tivoli Directory Integrator Assembly Line. 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 R/3 object. The Connector should be added to the **Flow** section of a Tivoli Directory Integrator Assembly Line. 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 Assembly Line 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 R/3 database, in turn, and make those details available to the Assembly Line. 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 Assembly Line 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 R/3 database. The Assembly Line 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 Assembly Line. 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());
  }
}
```

Troubleshooting the SAP R/3 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.

Missing `sapjco.jar`
If you attempt to use the SAP R/3 RFC FC and get an error similar to the following:

```
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 *TDI_Home*`/jars` directory. Refer to the instructions in "Configuring the SAP Java Connector" on page 306.

Missing `librfc32.dll`
If you attempt to use SAP R/3 RFC FC and get an error similar to the following:
"The dynamic linked library LIBRFC32.dll could not be found in the specified path"
On Windows machines, ensure that `librfc32.dll` is in the *TDI_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 Assembly Line

During testing, it was observed that changing the mode of Connector in the Assembly Line did not always work. The Connector sometimes appeared to execute in its original mode, resulting in Assembly Line errors. If this occurs, delete the Connector and add it to the Assembly Line 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 R/3 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 7. Script languages

With this version of IBM Tivoli Directory Integrator the only script language available is JavaSript, 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.

JavaScript

There are certain issues you might want to consider when using JavaScript. These are:

- "Comparing JavaScript strings" in *IBM Tivoli Directory Integrator 6.1: Users Guide*.
- "JavaScript string methods" in *IBM Tivoli Directory Integrator 6.1: 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 fails:

```
var myFile = java.io.File("file.txt"); myFile.delete();
```

Instead, you can do one of the following:

- `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.

© Copyright IBM Corp. 2002, 2006

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 **Help>Low Level API** 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).

© Copyright IBM Corp. 2002, 2006

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" on page 357.

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 and EventHandlers. 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 6.1: Users Guide*.

work The data container object of the AssemblyLine. It is therefore accessible in every Connector from the AssemblyLine.

event The **event** object is the **work** Entry object when scripting inside EventHandlers, like the **conn** object during Attribute Mapping.

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

See also

"The attribute object" on page 355.

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.logsmg ("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 6.1: Users Guide*.

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.
- For EventHandlers, this is the EventHandler thread where you can access EventHandler specific information and methods. See class `com.ibm.di.eventhandler.Switchboard` in the Javadocs.

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.

The following sections describe the specialized password synchronization components that are currently available.

Password Synchronizers

Password Synchronizer for Windows NT/2000/XP
Intercepts the Windows login password change.

Password Synchronizer for IBM Tivoli Directory Server
Intercepts IBM Tivoli Directory Server password changes.

Password Synchronizer for Sun ONE 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.

© Copyright IBM Corp. 2002, 2006

MQ Everyplace Password Store

Provides the function necessary to store user passwords into IBM WebSphere MQ Everyplace.

Note: IBM Tivoli Directory Integrator 6.1 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 and/or application submitting a change password request to the "MQe Password Store Connector" on page 137.

Specialized Connectors

MQe 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.

For more information about installing and configuring the IBM Password Synchronization Plug-ins, please see the *IBM Tivoli Directory Integrator 6.1: Password Synchronization Plug-ins Guide*.

Appendix B. AssemblyLine and Connector mode flowcharts

© Copyright IBM Corp. 2002, 2006

IBM Directory Integrator

Legend for Diagrams

Hook Flow diagrams

Black arrows indicate normal flow.

Red arrows show error/exception flow. Errors can occur both in scripted flow components, as well as in Integrator operations.

The Flow Endpoint symbol represents the start or end of the flow for a flow diagram. The text contained in the symbol provides more information about system state and behavior at this point.

These boxes represent scripted flow components, and are used for both Attribute Maps and Hooks. Note that if a Hook is enabled, then control is passed to the script in the Hook. If a Hook is not enabled, then the flow continues past the Hook without executing it.

A few Hooks are *mandatory* and must be enabled, although they do not need to actually contain any script. If a mandatory Hook is not enabled and the flow reaches this point, then this is considered an **error**, and control faults out to error handling.

This box represents an Integrator operation (these are available as functions in the component Interface object. Note that Integrator operations may also result in error flows.

Decision components represent logical branches in component flow execution, depending on state information at this point.

The yellow trapezoid describe flow which is detailed elsewhere (i.e. on another page in this document. The optional rounded blue box includes the

The Continuation symbol indicates that the flow is continued on another page that is common for one or more modes. The page being referenced will appear in a label below this symbol.

This is a Continuation symbol that is used when the referenced page is still part of the same component mode flow. The page being referenced will appear in a label below this symbol.

Hook Flow rev. 6.1
20060519

AssemblyLine flow

This flow diagram is for an AssemblyLine started by a main thread:

IBM Directory Integrator

AssemblyLine Flow

Hook Flow diagrams

*Flow References

These yellow trapezoids represent flows found in the AssemblyLine components.

Initialization Flows are found on the pages entitled **Initialization & Close Flows**

Iterator Flow is described on the page for **Iterator** Mode flow.

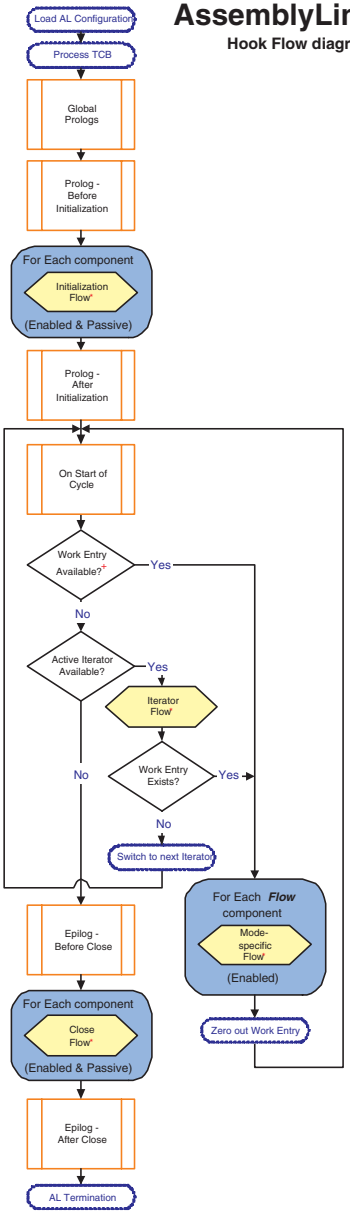
Mode-specific Flow can be found on the page(s) for that component Mode.

+Work Entry Available

This test checks to see if there is an **Entry** object which is to be used as **work** for the new cycle.

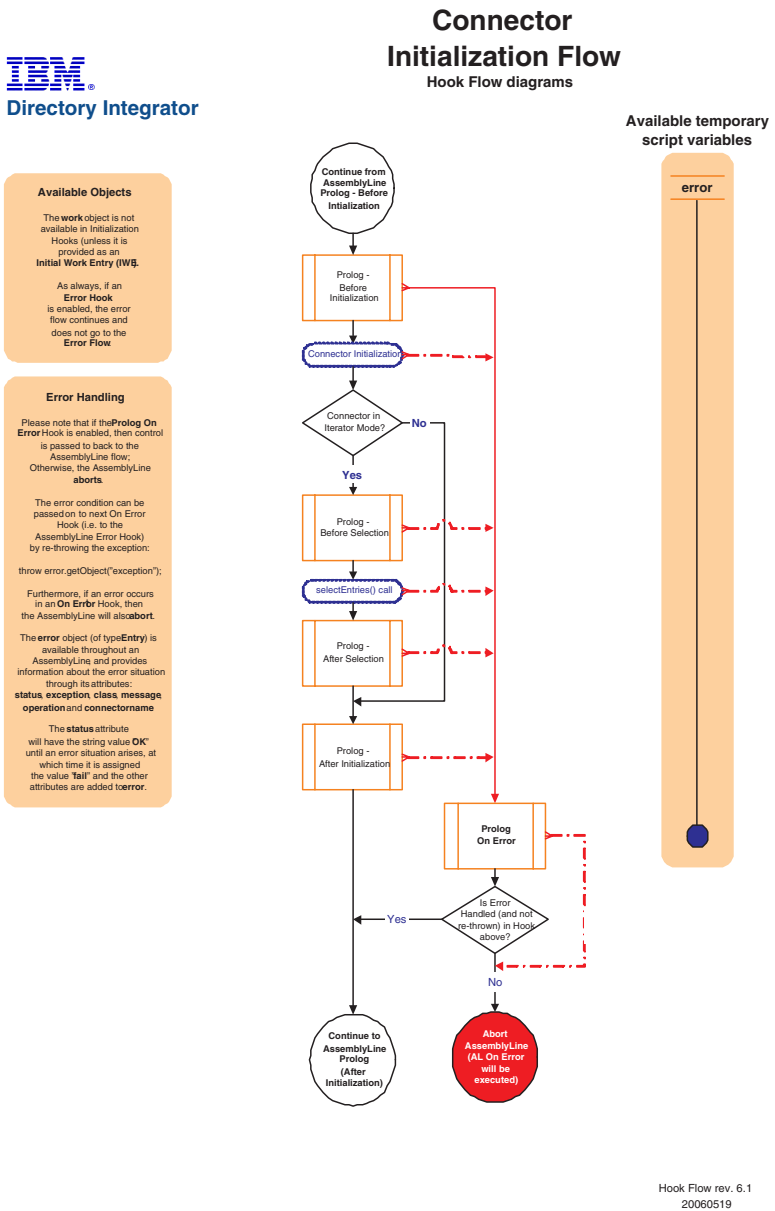
This Entry can be provided in a number of ways:

- o an **Initial Work Entry** (IWE)
- o via a call to **task.setWork()**
- o using **system.restartEntry()**

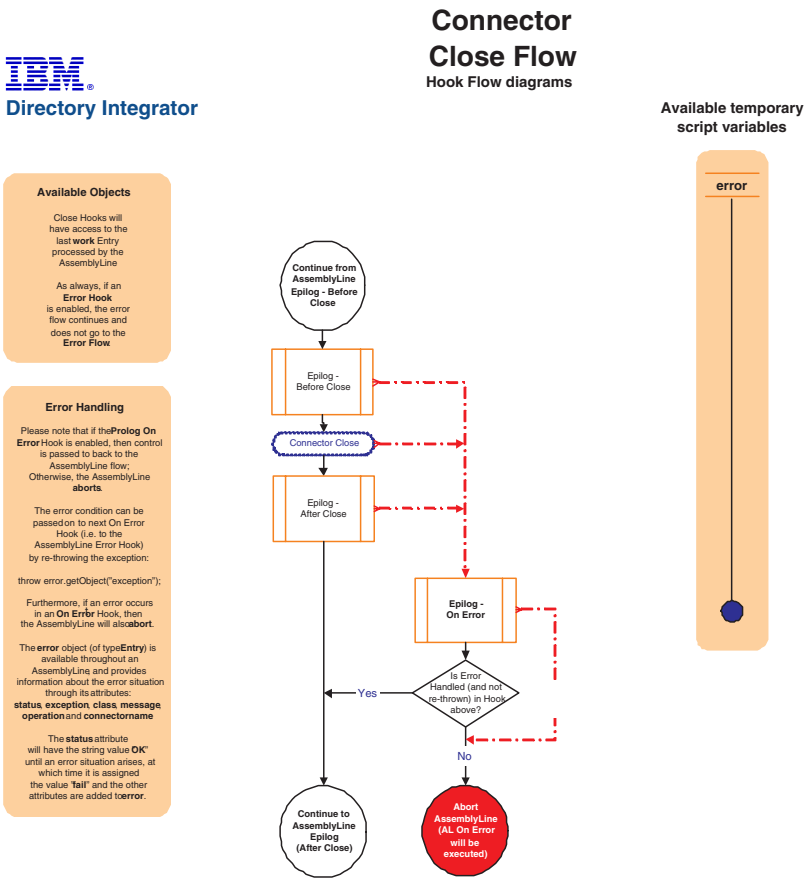


Hook Flow rev. 6.1
20060519

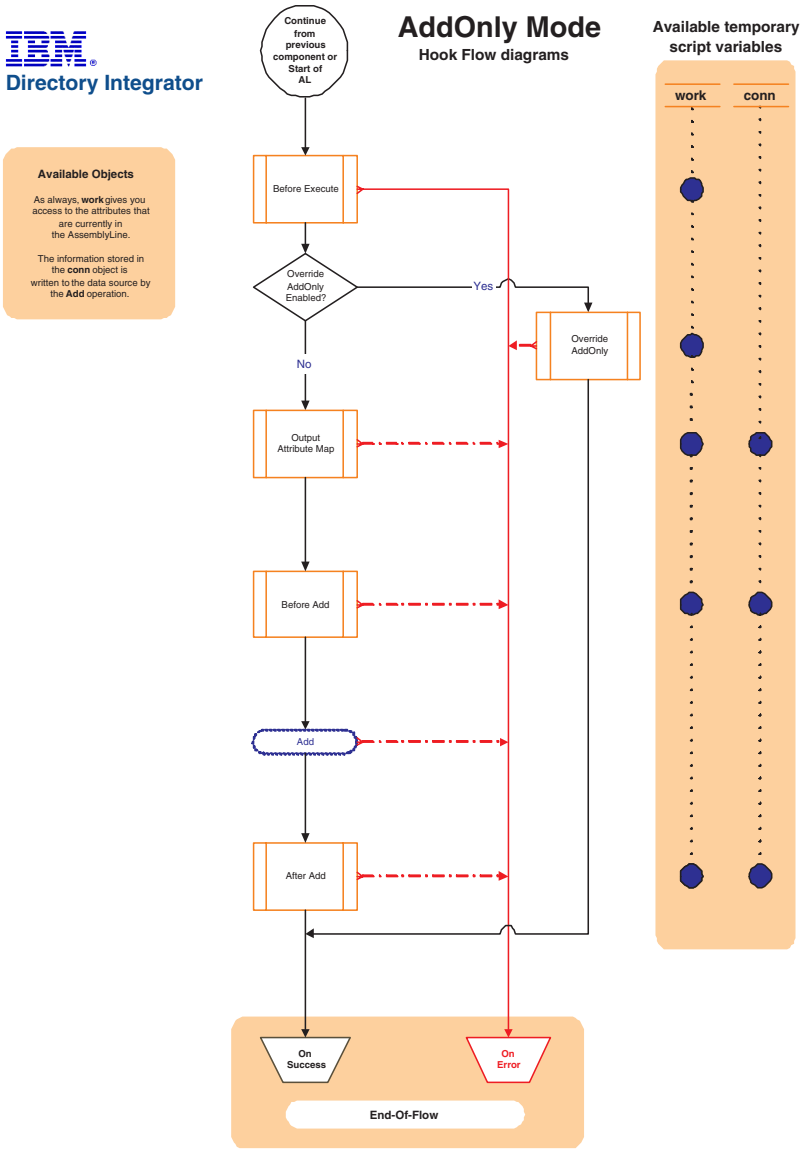
Connector initialization



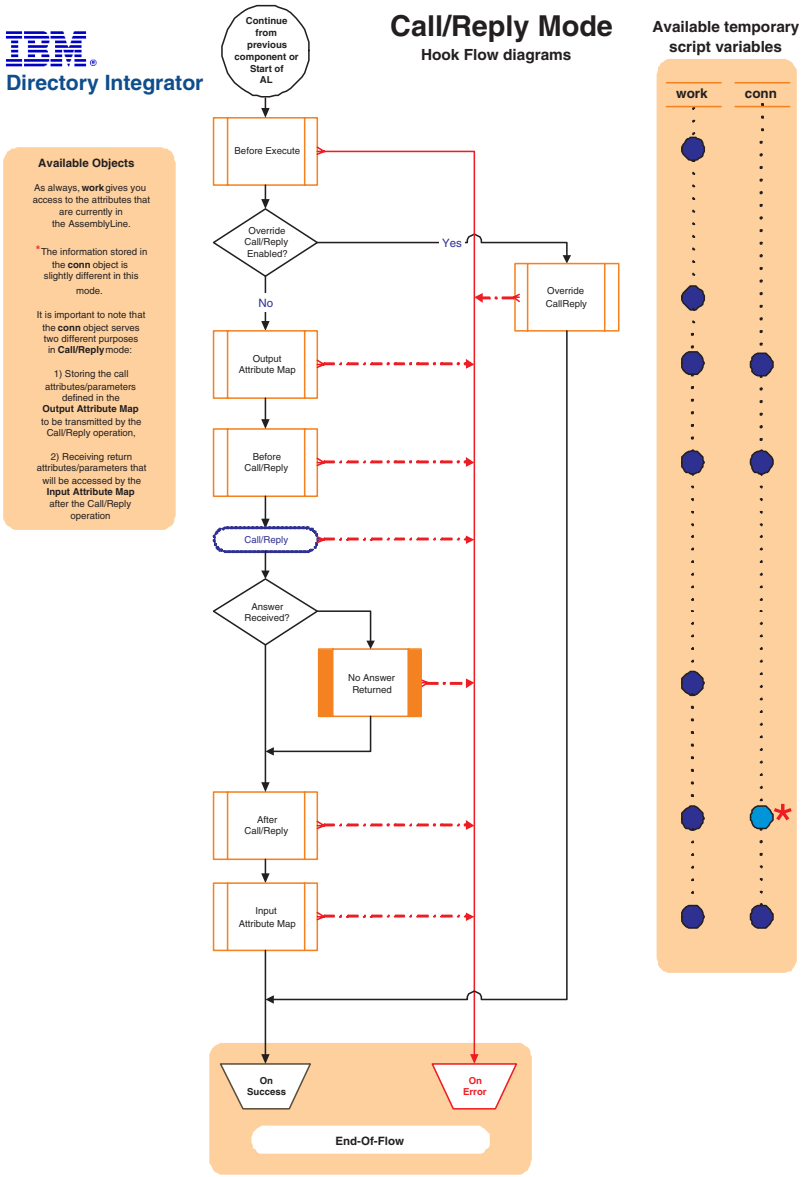
Close flow


Connector
Close Flow
Hook Flow diagrams
IBM
Directory Integrator
Available Objects
Close Hooks will have access to the last work Entry processed by the AssemblyLine
As always, if an Error Hook is enabled, the error flow continues and does not go to the Error Flow
Error Handling
Please note that if the Prolog On Error Hook is enabled, then control is passed to back to the AssemblyLine flow; Otherwise, the AssemblyLine aborts.
The error condition can be passed on to next On Error Hook (i.e. to the AssemblyLine Error Hook) by re-throwing the exception:
throw error.getObject("exception");
Furthermore, if an error occurs in an On Error Hook, then the AssemblyLine will also abort.
The error object (of type Entry) is available throughout an AssemblyLine and provides information about the error situation through its attributes: status, exception, class, message, operation and connectorname
The status attribute will have the string value OK" until an error situation arises, at which time it is assigned the value "fail" and the other attributes are added to error.
Continue from AssemblyLine Epilog - Before Close
Epilog - Before Close
Connector Close
Epilog - After Close
Epilog - On Error
Is Error Handled (and not re-thrown) in Hook above?
Yes
No
Continue to AssemblyLine Epilog (After Close)
Abort AssemblyLine (AL On Error will be executed)
Available temporary script variables
error
Hook Flow rev. 6.1
20060519

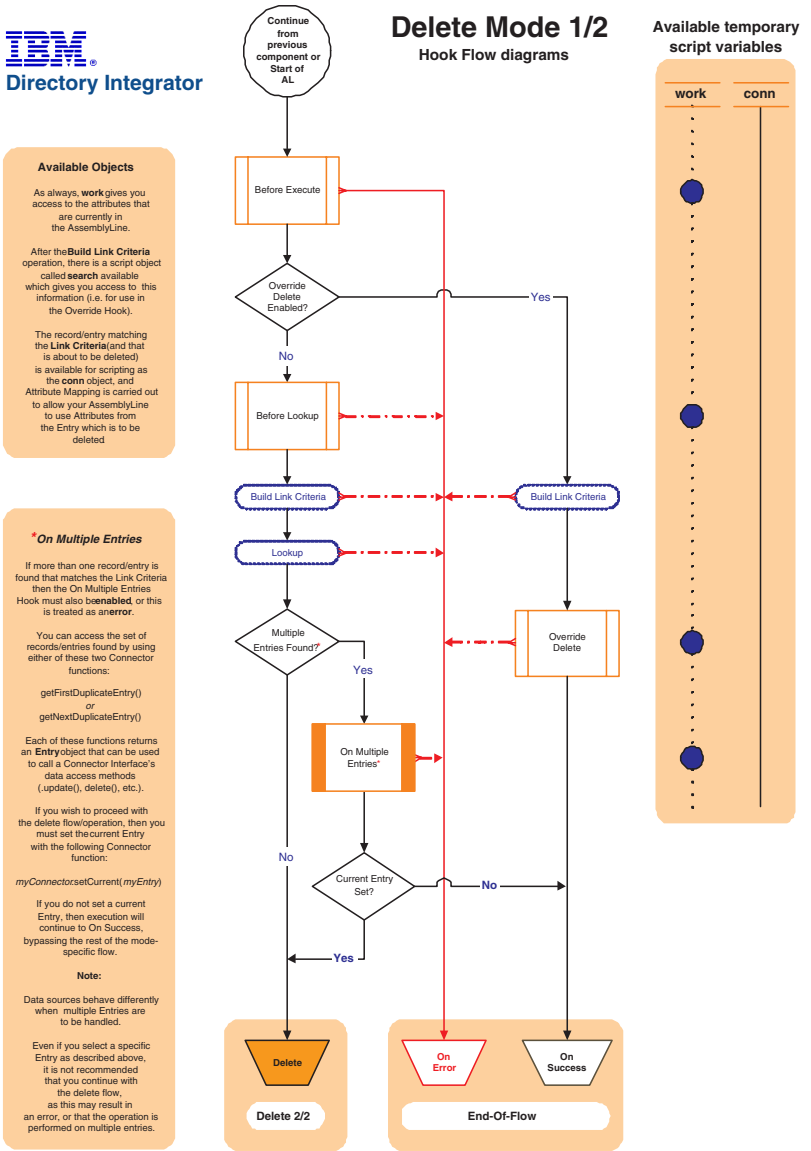
AddOnly mode

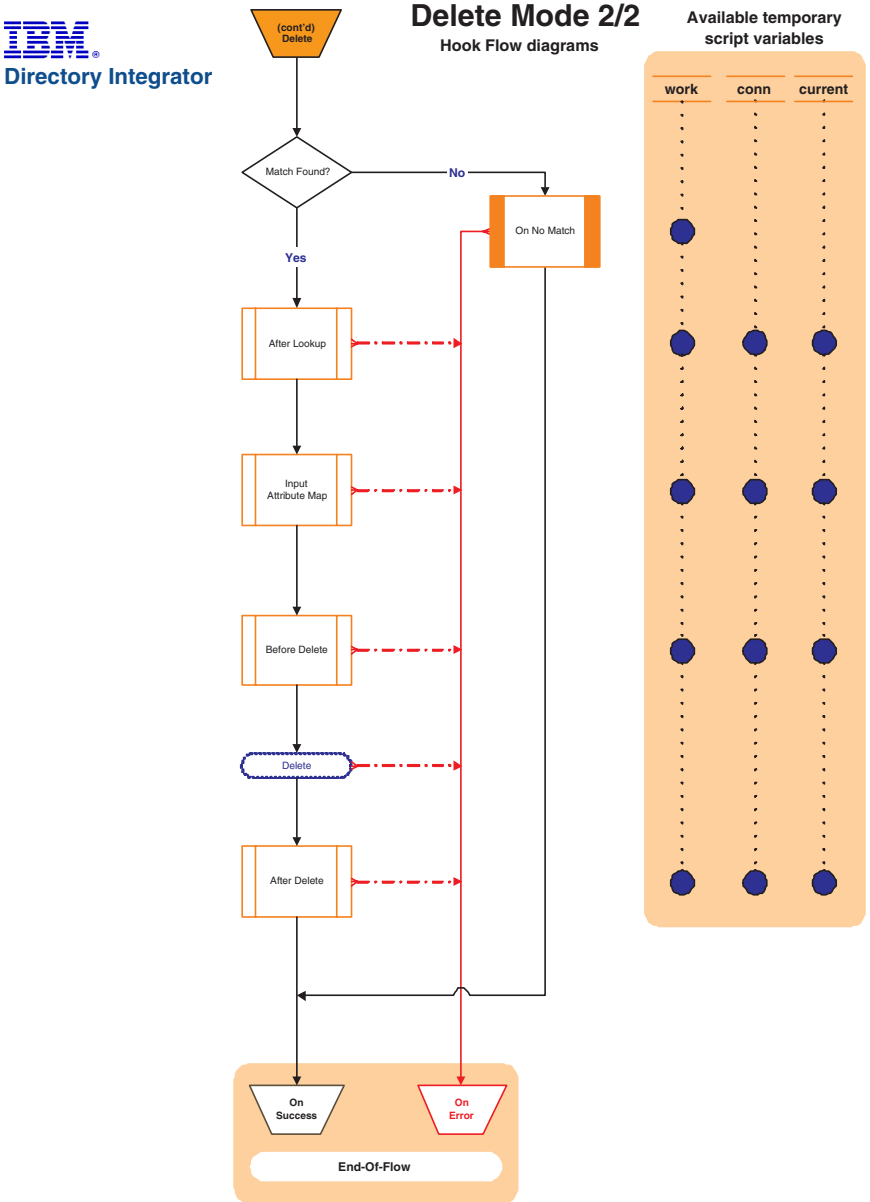


Call/Reply mode

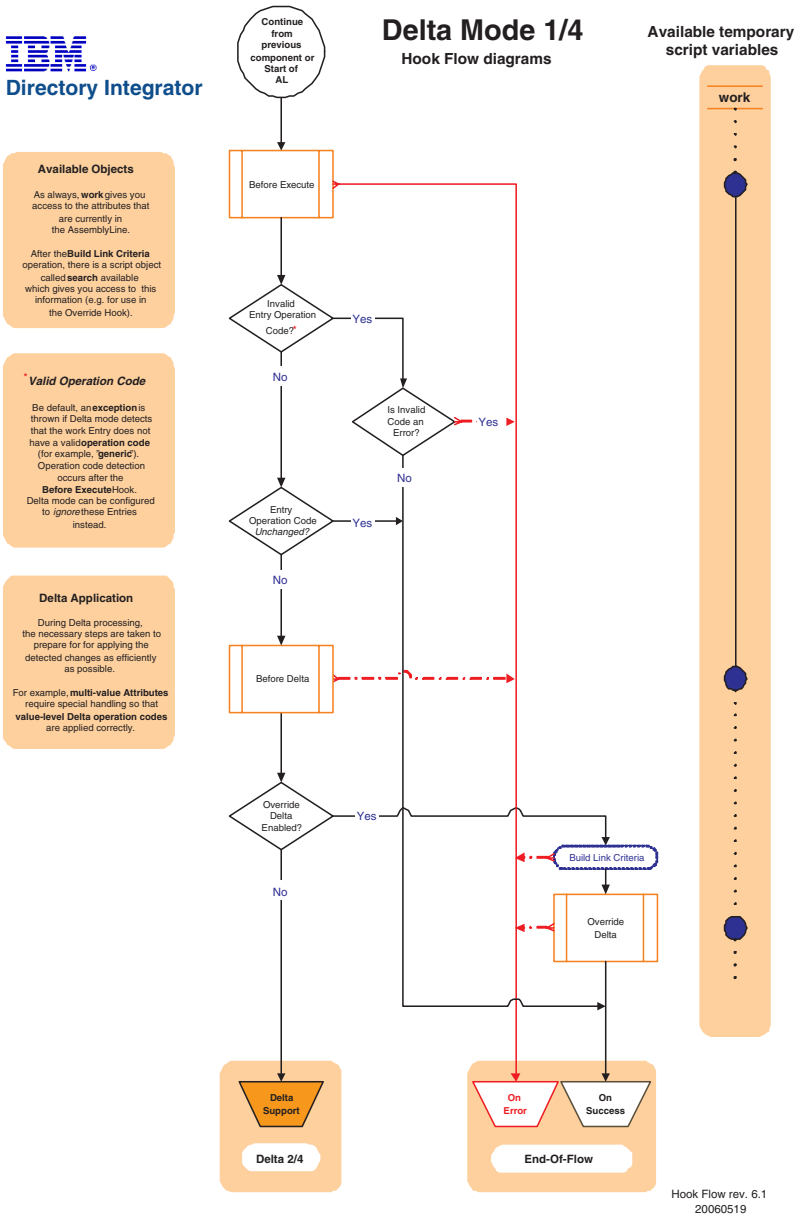


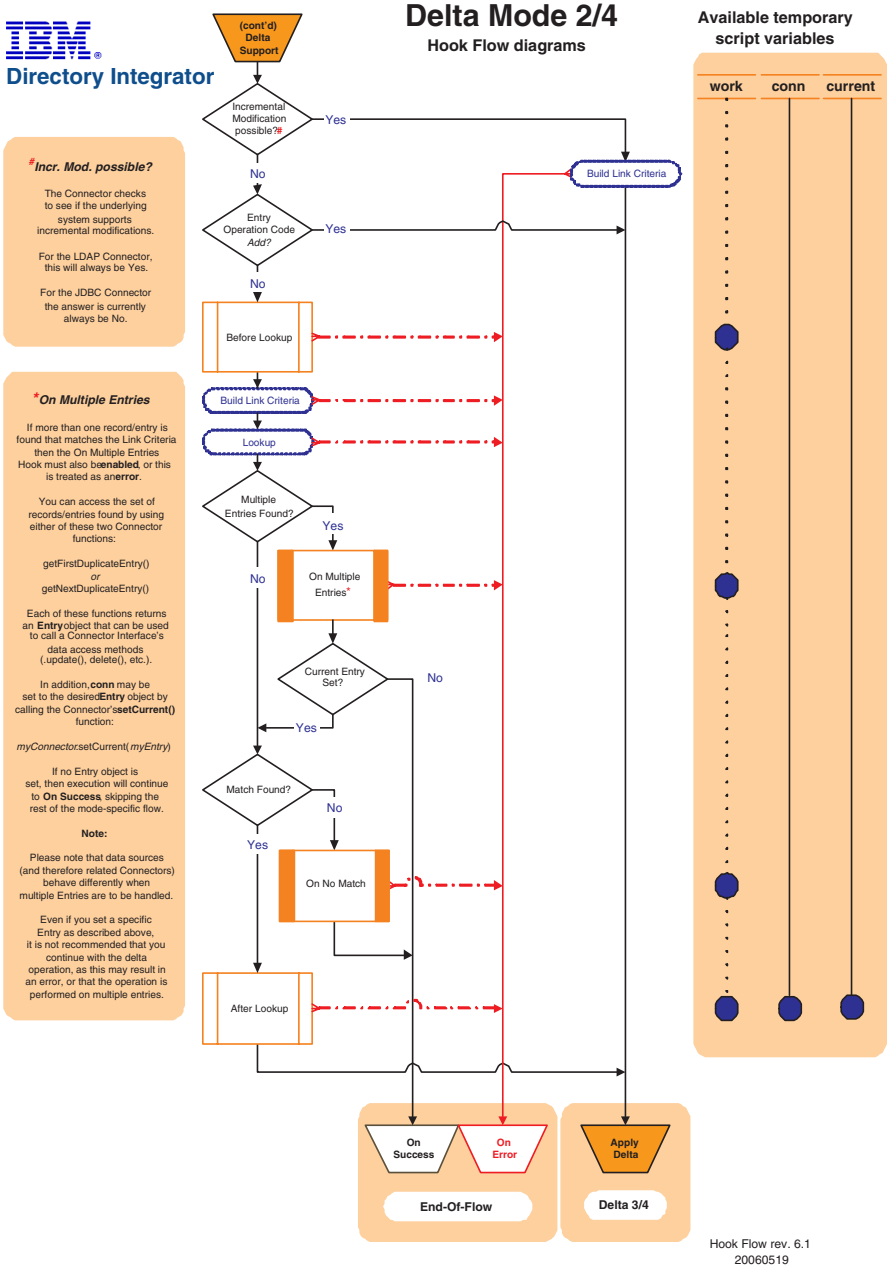
Delete mode

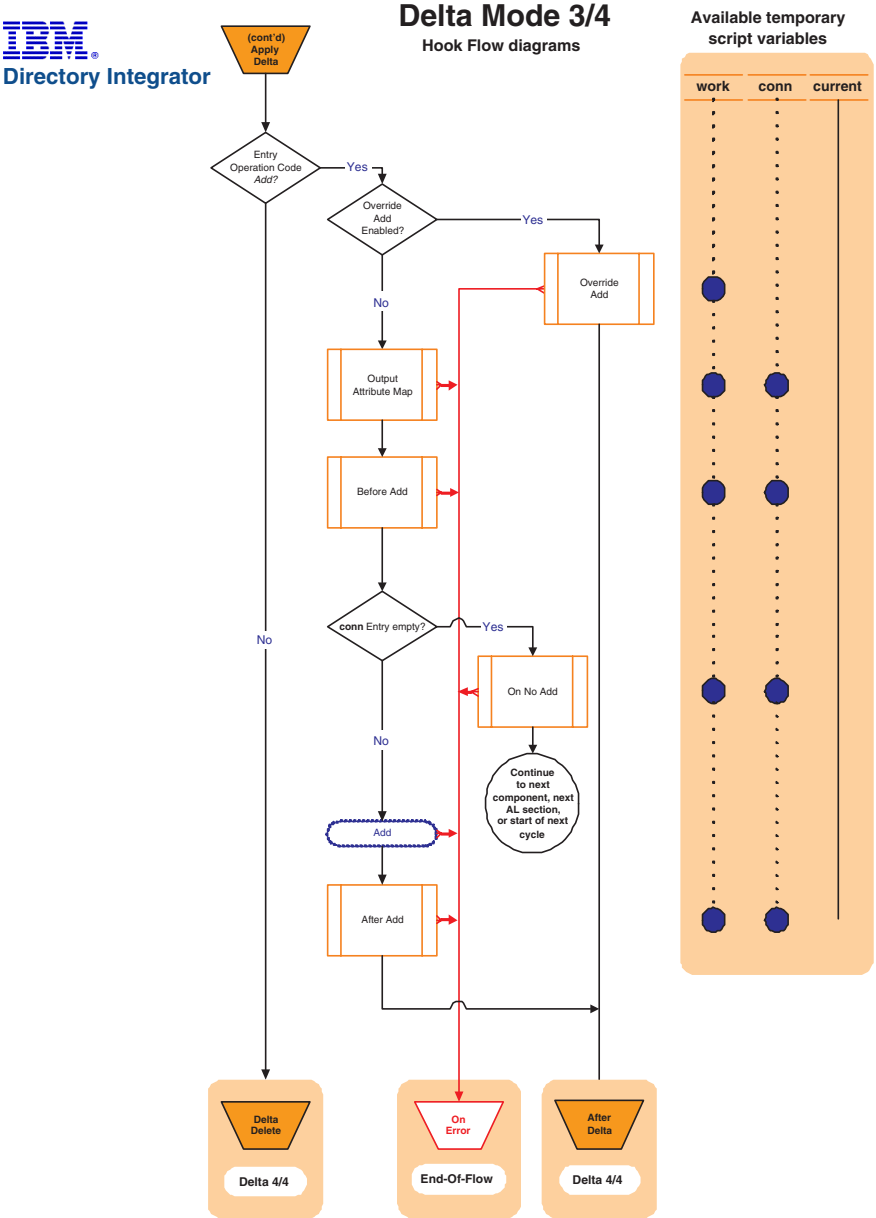

IBM
Directory Integrator
Delete Mode 1/2
Hook Flow diagrams
Available temporary script variables
work
conn
Continue from previous component or Start of AL
Before Execute
Override Delete Enabled?
Yes
No
Before Lookup
Build Link Criteria
Build Link Criteria
Lookup
Multiple Entries Found?*
Yes
Override Delete
On Multiple Entries*
No
Current Entry Set?
No
Yes
Delete
Delete 2/2
On Error
On Success
End-Of-Flow
Available Objects
As always, work gives you access to the attributes that are currently in the AssemblyLine.
After the Build Link Criteria operation, there is a script object called search available which gives you access to this information (i.e. for use in the Override Hook).
The record/entry matching the Link Criteria (and that is about to be deleted) is available for scripting as the conn object, and Attribute Mapping is carried out to allow your AssemblyLine to use Attributes from the Entry which is to be deleted
*On Multiple Entries
If more than one record/entry is found that matches the Link Criteria then the On Multiple Entries Hook must also be enabled, or this is treated as an error.
You can access the set of records/entries found by using either of these two Connector functions:
getFirstDuplicateEntry()
or
getNextDuplicateEntry()
Each of these functions returns an Entry object that can be used to call a Connector Interface's data access methods (.update(), delete(), etc.).
If you wish to proceed with the delete flow/operation, then you must set the current Entry with the following Connector function:
myConnector.setCurrent(myEntry)
If you do not set a current Entry, then execution will continue to On Success, bypassing the rest of the mode-specific flow.
Note:
Data sources behave differently when multiple Entries are to be handled.
Even if you select a specific Entry as described above, it is not recommended that you continue with the delete flow, as this may result in an error, or that the operation is performed on multiple entries.
Hook Flow rev. 6.1
20060519


IBM
Directory Integrator
Delete Mode 2/2
Hook Flow diagrams
(cont'd) Delete
Match Found?
No
Yes
On No Match
After Lookup
Input Attribute Map
Before Delete
Delete
After Delete
On Success
On Error
End-Of-Flow
Available temporary script variables
work
conn
current
Hook Flow rev. 6.1
20060519

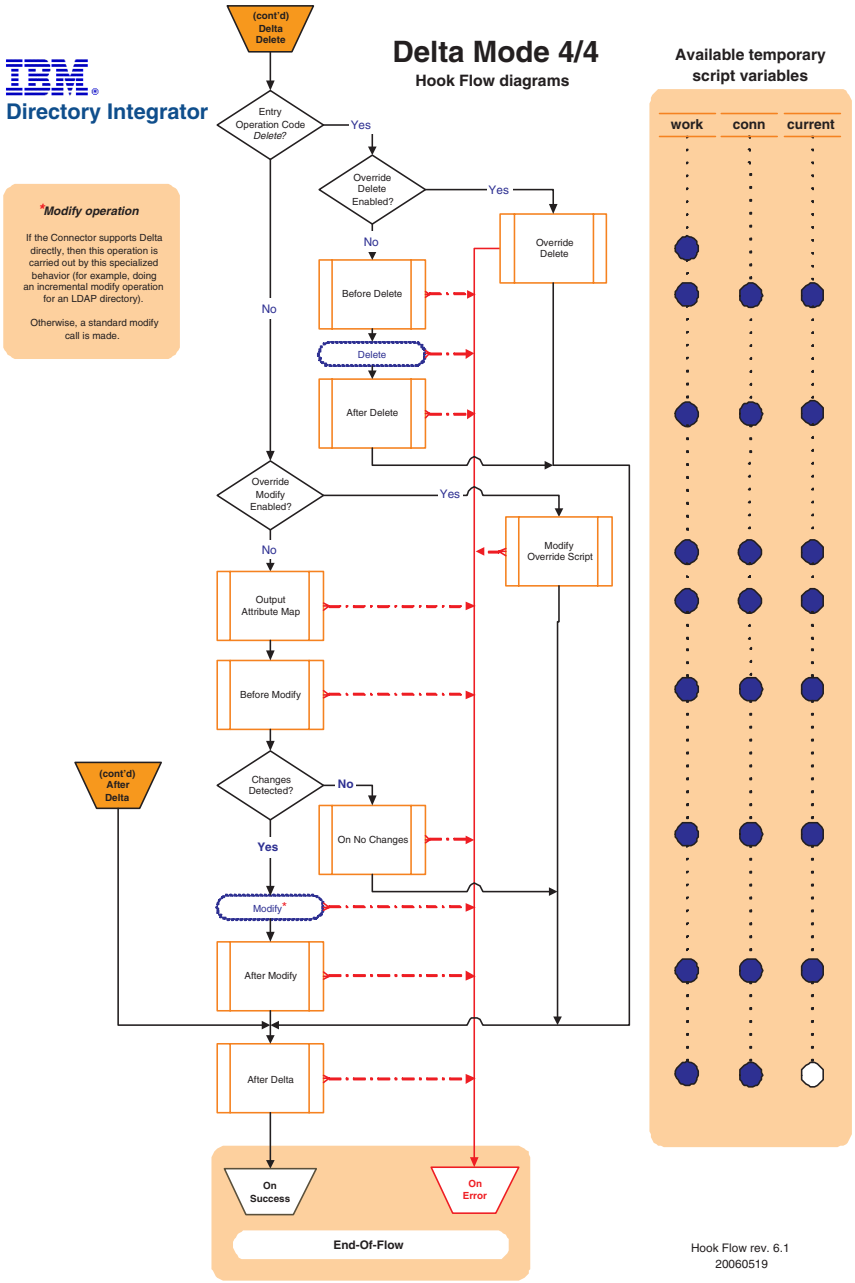
Delta Mode


IBM
Directory Integrator
Delta Mode 1/4
Hook Flow diagrams
Available temporary script variables
work
Continue from previous component or Start of AL
Before Execute
Invalid Entry Operation Code?*
Yes
No
Is Invalid Code an Error?
Yes
No
Entry Operation Code Unchanged?
Yes
No
Before Delta
Override Delta Enabled?
Yes
No
Build Link Criteria
Override Delta
Delta Support
Delta 2/4
On Error
On Success
End-Of-Flow
Available Objects
As always, work gives you access to the attributes that are currently in the AssemblyLine.
After the Build Link Criteria operation, there is a script object called search available which gives you access to this information (e.g. for use in the Override Hook).
* Valid Operation Code
Be default, an exception is thrown if Delta mode detects that the work Entry does not have a valid operation code (for example, 'generic'). Operation code detection occurs after the Before Execute Hook. Delta mode can be configured to ignore these Entries instead.
Delta Application
During Delta processing, the necessary steps are taken to prepare for for applying the detected changes as efficiently as possible.
For example, multi-value Attributes require special handling so that value-level Delta operation codes are applied correctly.
Hook Flow rev. 6.1
20060519

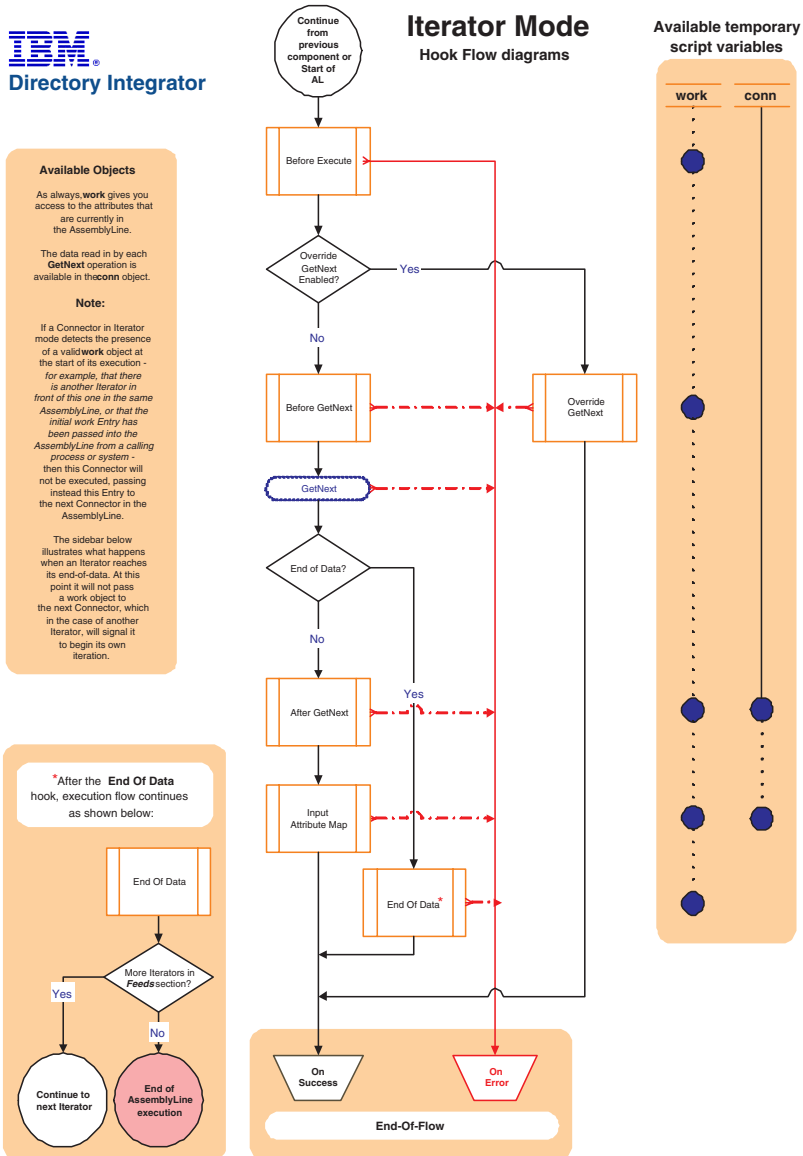

IBM
Directory Integrator
Delta Mode 2/4
Hook Flow diagrams
(cont'd) Delta Support
Incremental Modification possible?#
Yes
No
Entry Operation Code Add?
Yes
No
Build Link Criteria
Before Lookup
Build Link Criteria
Lookup
Multiple Entries Found?
Yes
No
On Multiple Entries*
Current Entry Set?
No
Yes
Match Found?
No
Yes
On No Match
After Lookup
On Success
On Error
End-Of-Flow
Apply Delta
Delta 3/4
#Incr. Mod. possible?
The Connector checks to see if the underlying system supports incremental modifications.
For the LDAP Connector, this will always be Yes.
For the JDBC Connector the answer is currently always be No.
*On Multiple Entries
If more than one record/entry is found that matches the Link Criteria then the On Multiple Entries Hook must also be enabled, or this is treated as an error.
You can access the set of records/entries found by using either of these two Connector functions:
getFirstDuplicateEntry()
or
getNextDuplicateEntry()
Each of these functions returns an Entry object that can be used to call a Connector Interface's data access methods (.update(), delete(), etc.).
In addition, conn may be set to the desired Entry object by calling the Connector's setCurrent() function:
myConnector.setCurrent(myEntry)
If no Entry object is set, then execution will continue to On Success, skipping the rest of the mode-specific flow.
Note:
Please note that data sources (and therefore related Connectors) behave differently when multiple Entries are to be handled.
Even if you set a specific Entry as described above, it is not recommended that you continue with the delta operation, as this may result in an error, or that the operation is performed on multiple entries.
Available temporary script variables
work
conn
current
Hook Flow rev. 6.1
20060519



Hook Flow rev. 6.1
20060519

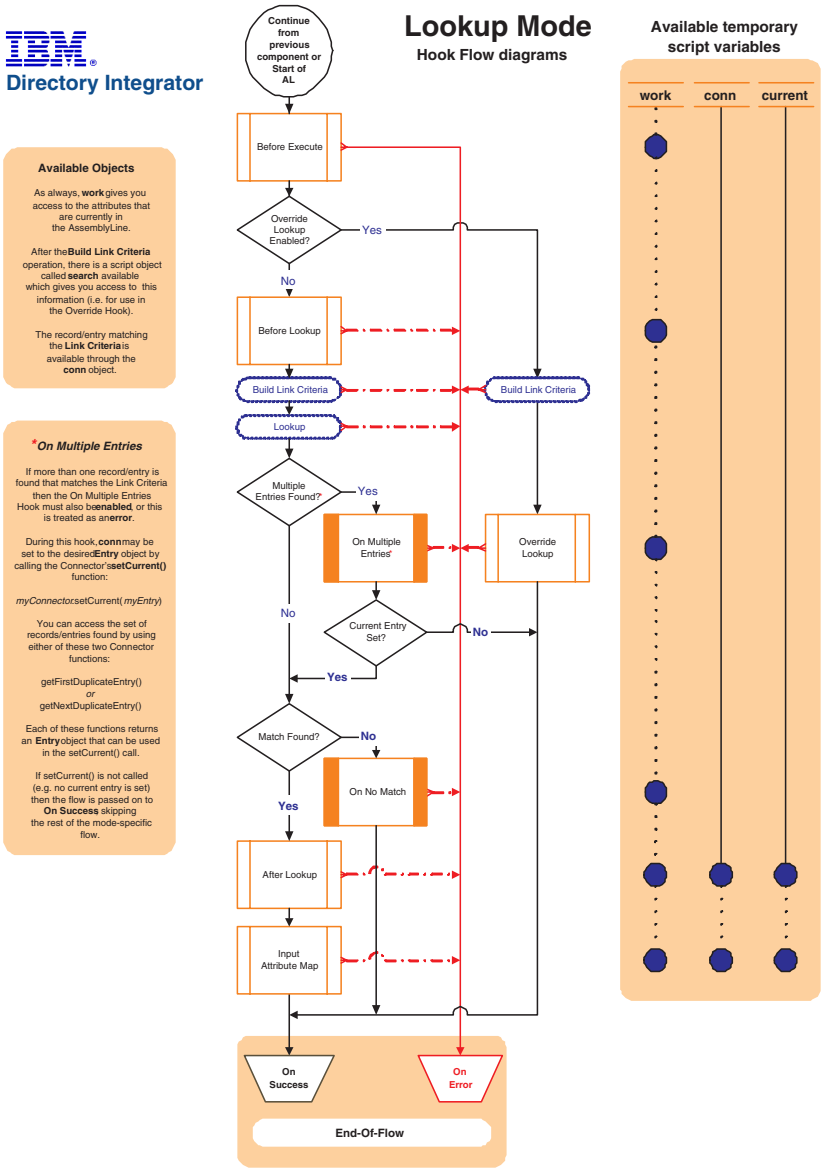

IBM
Directory Integrator
Delta Mode 4/4
Hook Flow diagrams
(cont'd) Delta Delete
Entry Operation Code Delete?
Yes
No
Override Delete Enabled?
Yes
No
Override Delete
Before Delete
Delete
After Delete
Override Modify Enabled?
Yes
No
Modify Override Script
Output Attribute Map
Before Modify
(cont'd) After Delta
Changes Detected?
No
Yes
On No Changes
Modify*
After Modify
After Delta
On Success
On Error
End-Of-Flow
*Modify operation
If the Connector supports Delta directly, then this operation is carried out by this specialized behavior (for example, doing an incremental modify operation for an LDAP directory).
Otherwise, a standard modify call is made.
Available temporary script variables
work
conn
current
Hook Flow rev. 6.1
20060519

Iterator mode

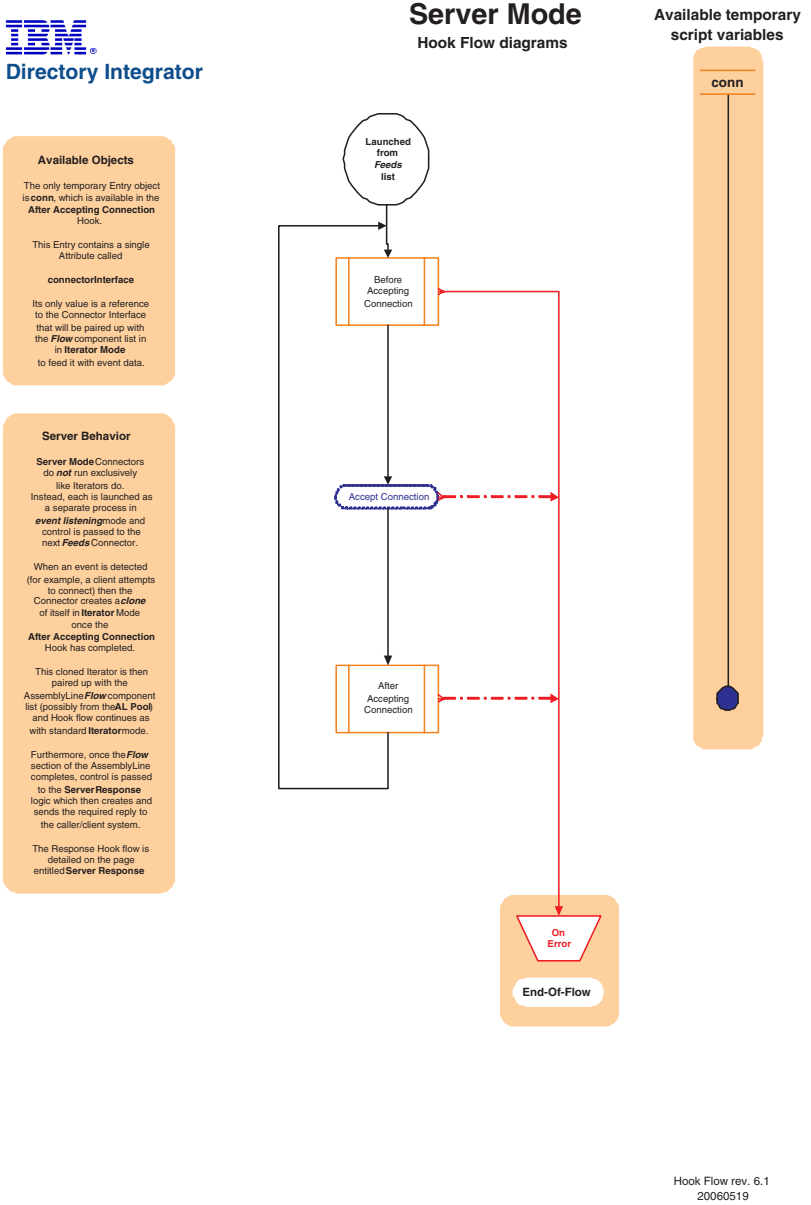


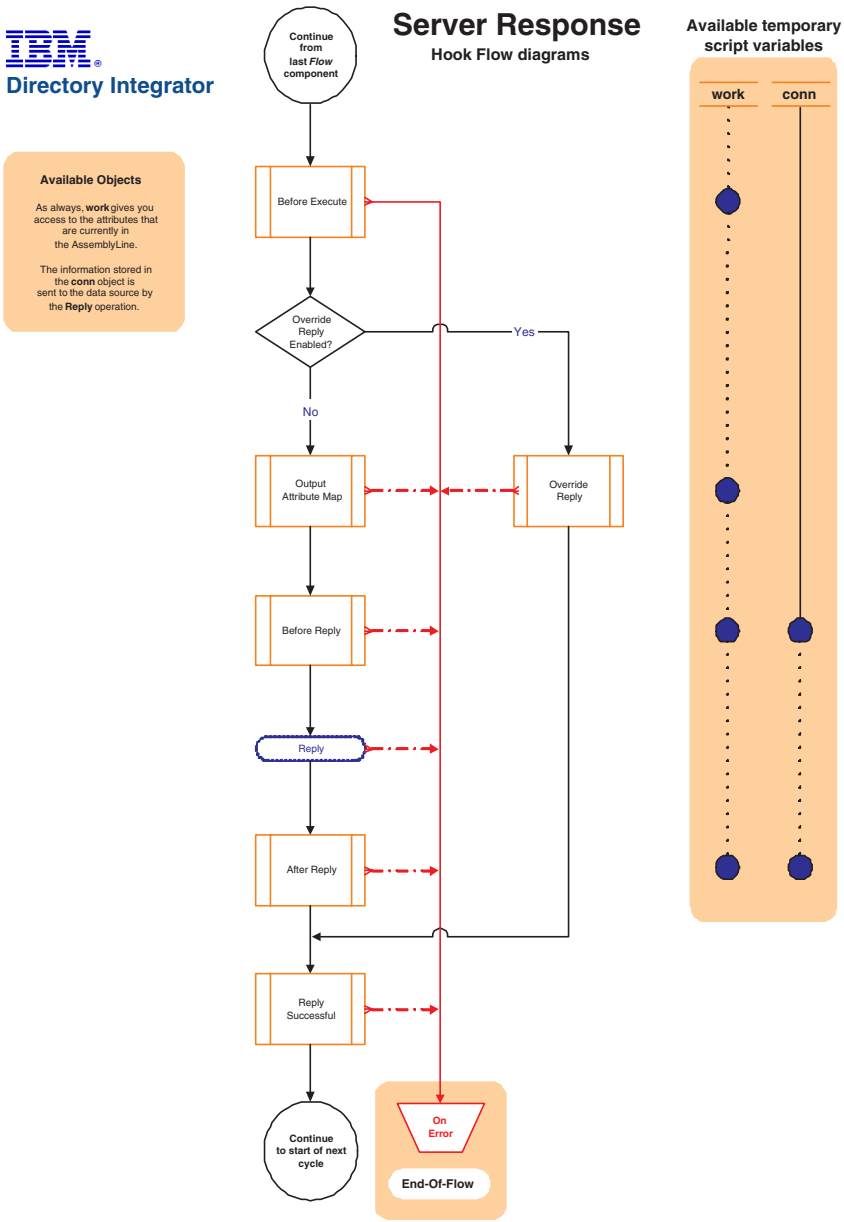
Hook Flow rev. 6.1
20060519

Lookup mode

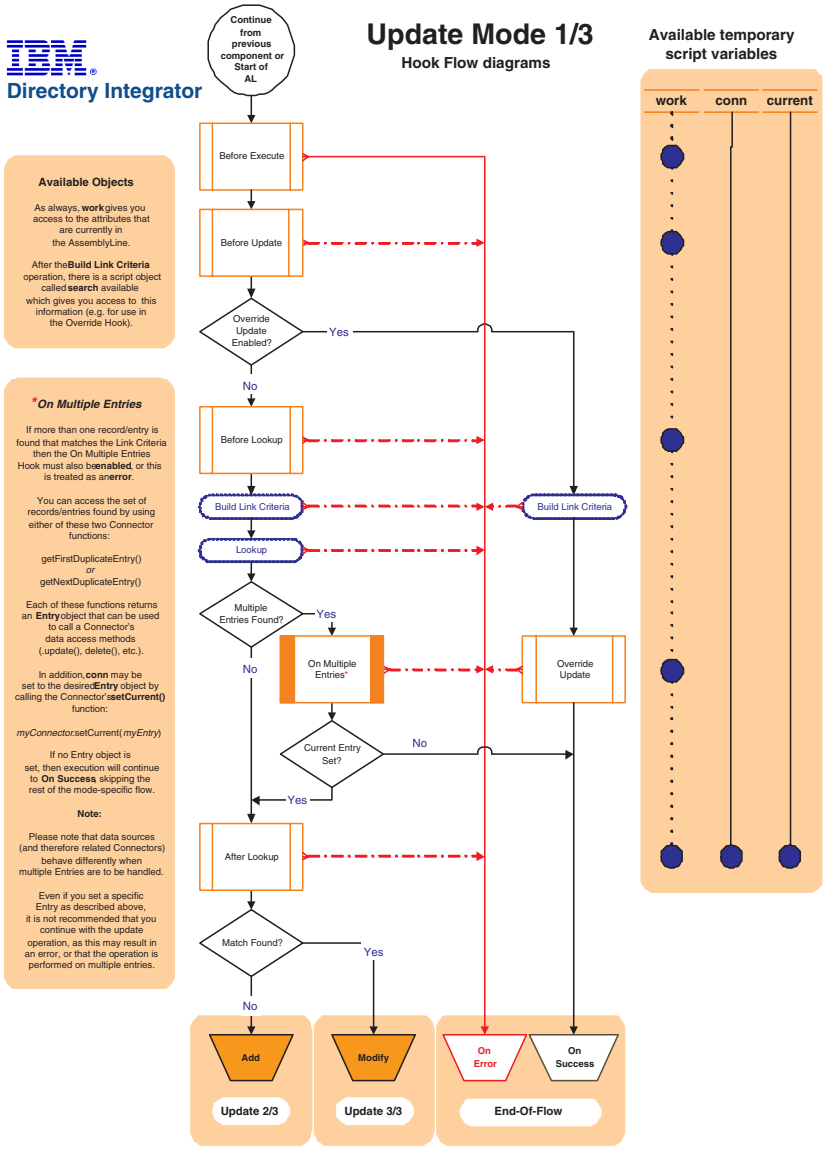

IBM
Directory Integrator
Available Objects
As always, work gives you access to the attributes that are currently in the AssemblyLine.
After the Build Link Criteria operation, there is a script object called search available which gives you access to this information (i.e. for use in the Override Hook).
The record/entry matching the Link Criteria is available through the conn object.
*On Multiple Entries
If more than one record/entry is found that matches the Link Criteria then the On Multiple Entries Hook must also be enabled, or this is treated as an error.
During this hook, conn may be set to the desired Entry object by calling the Connector's setCurrent() function:
myConnector.setCurrent(myEntry)
You can access the set of records/entries found by using either of these two Connector functions:
getFirstDuplicateEntry()
or
getNextDuplicateEntry()
Each of these functions returns an Entry object that can be used in the setCurrent() call.
If setCurrent() is not called (e.g. no current entry is set) then the flow is passed on to On Success skipping the rest of the mode-specific flow.
Lookup Mode
Hook Flow diagrams
Continue from previous component or Start of AL
Before Execute
Override Lookup Enabled?
Yes
No
Before Lookup
Build Link Criteria
Lookup
Multiple Entries Found?
Yes
On Multiple Entries*
No
Current Entry Set?
No
Yes
Match Found?
No
On No Match
Yes
After Lookup
Input Attribute Map
On Success
On Error
End-Of-Flow
Build Link Criteria
Override Lookup
Available temporary script variables
work
conn
current
Hook Flow rev. 6.1
20060519

Server Mode

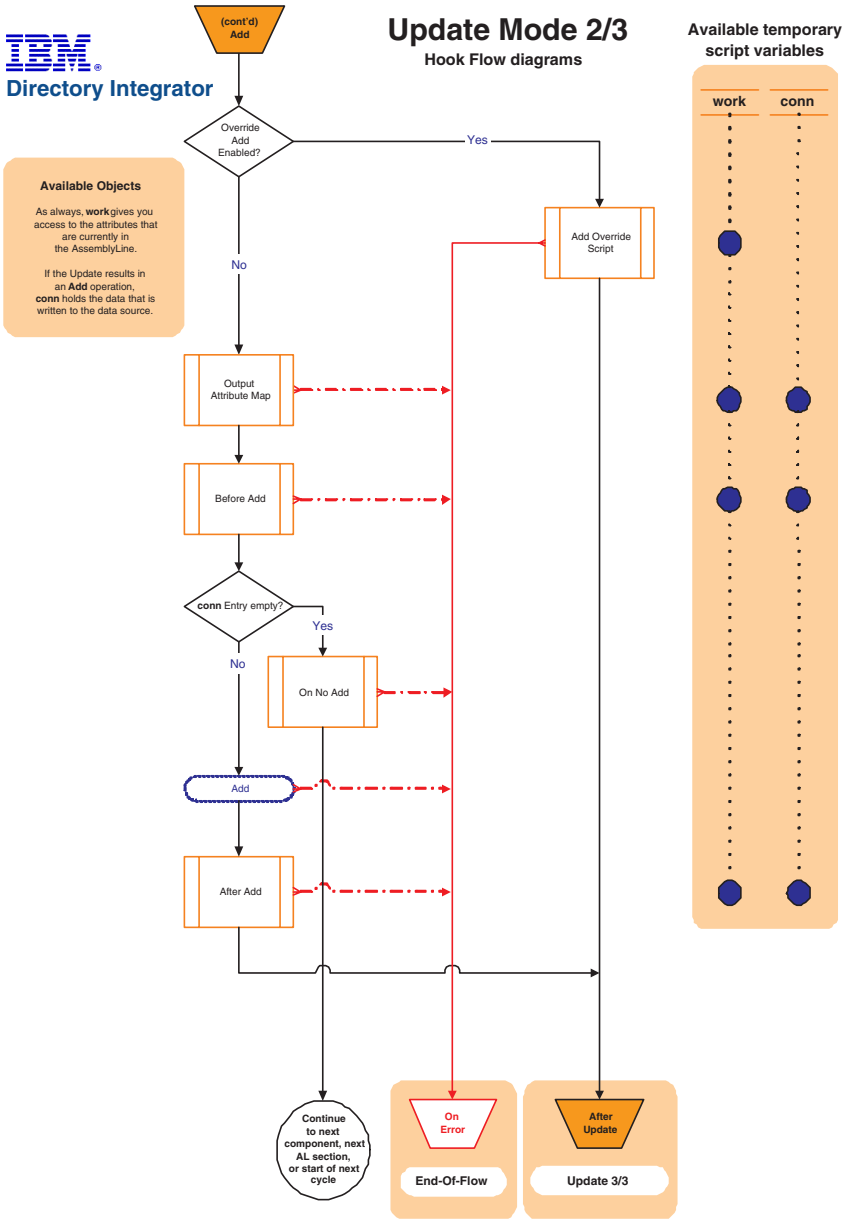

IBM
Directory Integrator
Server Mode
Hook Flow diagrams
Available Objects
The only temporary Entry object is conn, which is available in the After Accepting Connection Hook.
This Entry contains a single Attribute called
connectorInterface
Its only value is a reference to the Connector Interface that will be paired up with the Flow component list in in Iterator Mode to feed it with event data.
Server Behavior
Server Mode Connectors do not run exclusively like Iterators do. Instead, each is launched as a separate process in event listening mode and control is passed to the next Feeds Connector.
When an event is detected (for example, a client attempts to connect) then the Connector creates a clone of itself in Iterator Mode once the After Accepting Connection Hook has completed.
This cloned Iterator is then paired up with the AssemblyLine Flow component list (possibly from the AL Pool) and Hook flow continues as with standard Iterator mode.
Furthermore, once the Flow section of the AssemblyLine completes, control is passed to the Server Response logic which then creates and sends the required reply to the caller/client system.
The Response Hook flow is detailed on the page entitled Server Response
Launched from Feeds list
Before Accepting Connection
Accept Connection
After Accepting Connection
On Error
End-Of-Flow
Available temporary script variables
conn
Hook Flow rev. 6.1
20060519

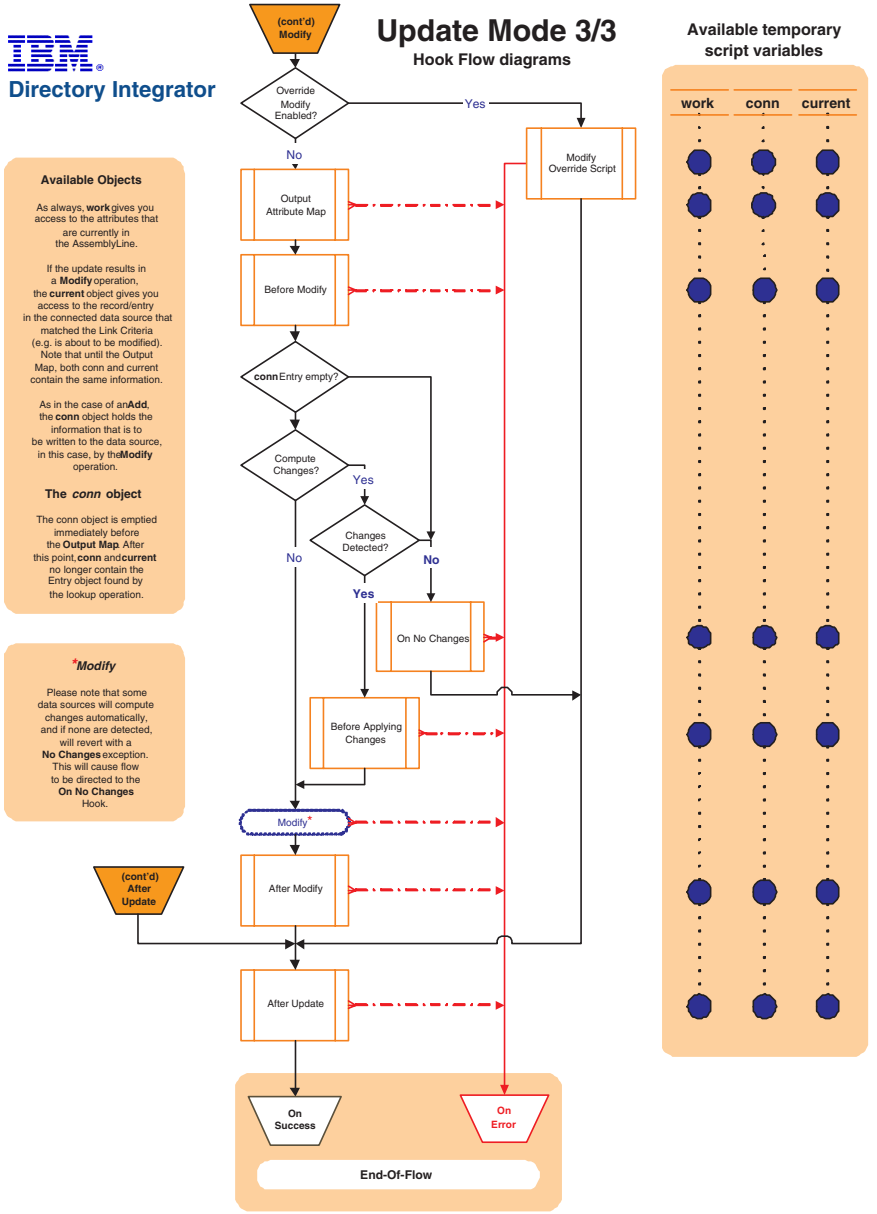

IBM
Directory Integrator
Server Response
Hook Flow diagrams
Available Objects
As always, work gives you access to the attributes that are currently in the AssemblyLine.
The information stored in the conn object is sent to the data source by the Reply operation.
Continue from last Flow component
Before Execute
Override Reply Enabled?
Yes
No
Output Attribute Map
Override Reply
Before Reply
Reply
After Reply
Reply Successful
Continue to start of next cycle
On Error
End-Of-Flow
Available temporary script variables
work
conn
Hook Flow rev. 6.1
20060519

Update mode



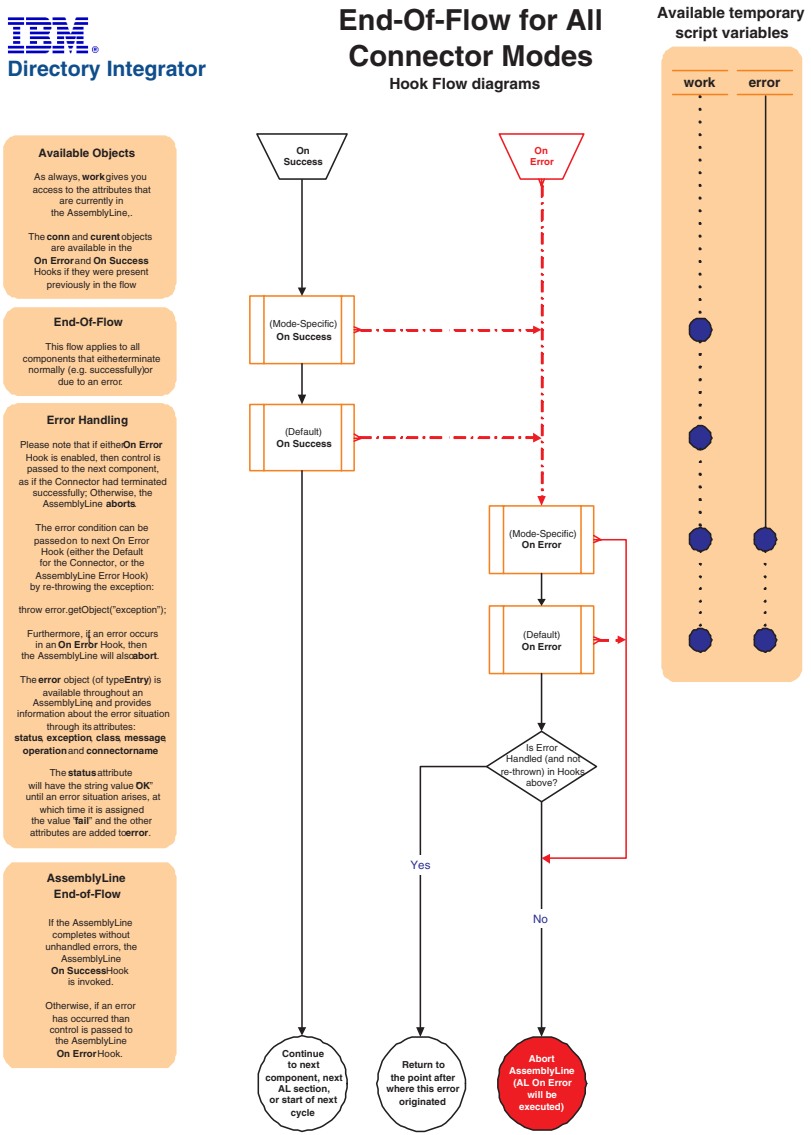
Hook Flow rev. 6.1
20060519


IBM
Directory Integrator
Update Mode 2/3
Hook Flow diagrams
(cont'd) Add
Override Add Enabled?
Yes
No
Add Override Script
Available Objects
As always, work gives you access to the attributes that are currently in the AssemblyLine.
If the Update results in an Add operation, conn holds the data that is written to the data source.
Output Attribute Map
Before Add
conn Entry empty?
Yes
No
On No Add
Add
After Add
Available temporary script variables
work
conn
Continue to next component, next AL section, or start of next cycle
On Error
End-Of-Flow
After Update
Update 3/3
Hook Flow rev. 6.1
20060519



Hook Flow rev. 6.1
20060519

End-of-flow for all modes

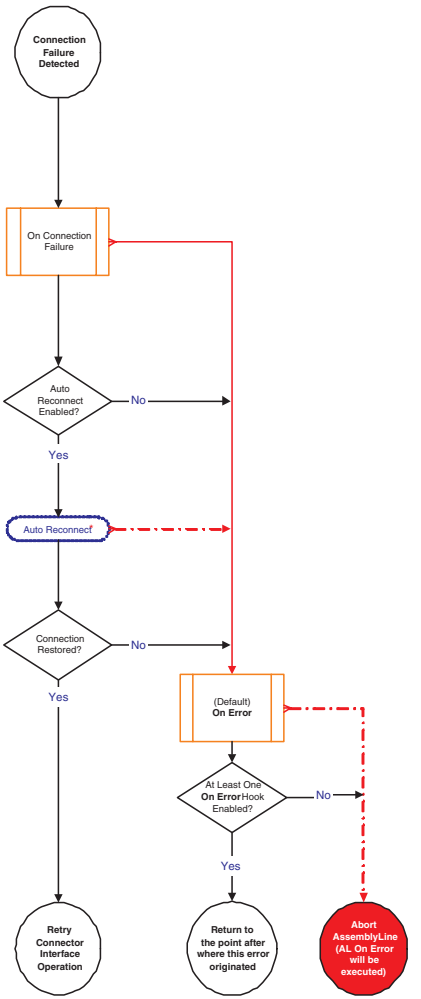


Hook Flow rev. 6.1
20060519

IBM Directory Integrator

Connector Reconnect

Hook Flow diagrams



Available Objects

As always, **work** gives you access to the attributes that are currently in the AssemblyLine,.

The **error** object (of type **Entry**) is available throughout an AssemblyLine, and provides information about the an error situation through its attributes: ***status***, ***exception class***, ***message***, ***operation*** and ***connectorname***.

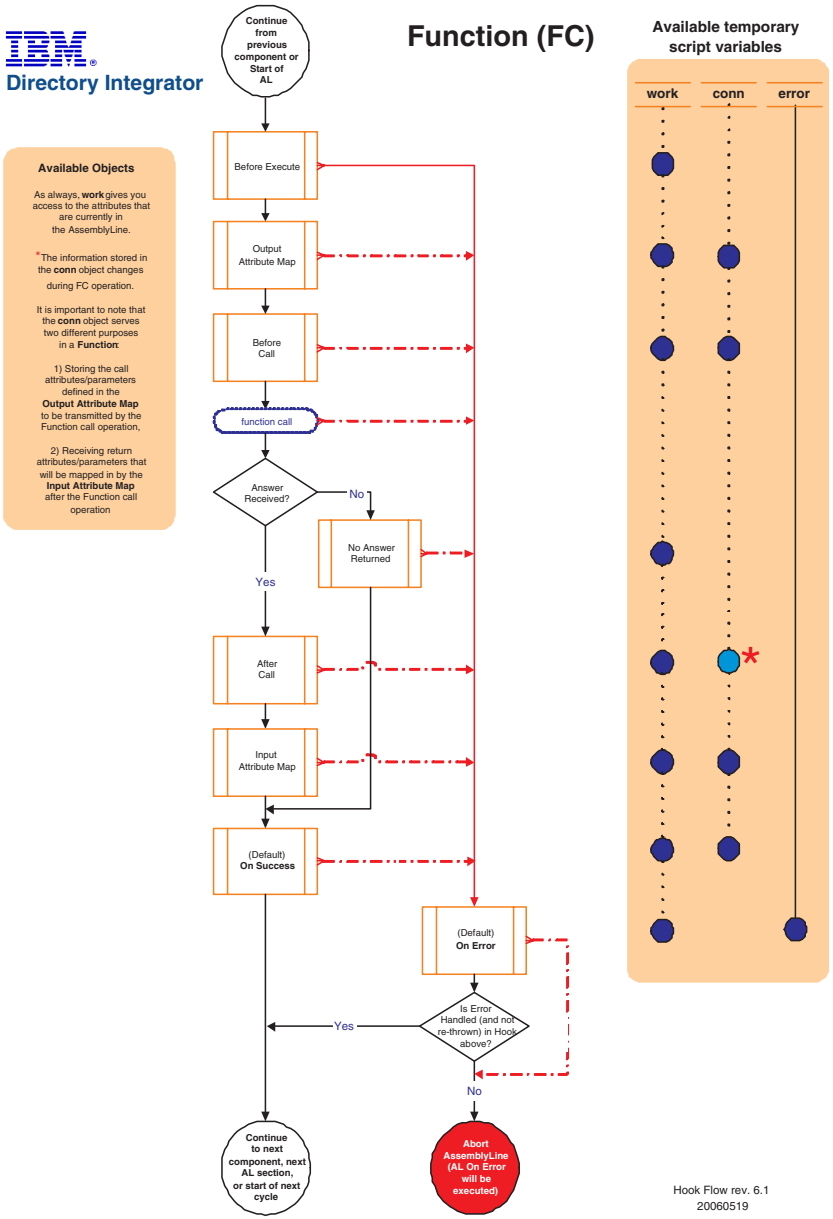
The **status** attribute will have the string value **OK**" until an error situation arises, at which time it is assigned the value '**fail**' and the other attributes are added to **error**.

Auto Reconnect

The Auto Reconnect feature is configured through the parameters found in the Connector **Reconnect** tab.

These parameters control the maximum number of times a reconnect will be tried, as well as the number seconds to wait between each attempt.

Hook Flow rev. 6.1
20060519


IBM
Directory Integrator
Function (FC)
Available Objects
As always, work gives you access to the attributes that are currently in the AssemblyLine.
*The information stored in the conn object changes during FC operation.
It is important to note that the conn object serves two different purposes in a Function:
1) Storing the call attributes/parameters defined in the Output Attribute Map to be transmitted by the Function call operation,
2) Receiving return attributes/parameters that will be mapped in by the Input Attribute Map after the Function call operation
Continue from previous component or Start of AL
Before Execute
Output Attribute Map
Before Call
function call
Answer Received?
No
No Answer Returned
Yes
After Call
Input Attribute Map
(Default) On Success
(Default) On Error
Is Error Handled (and not re-thrown) in Hook above?
Yes
No
Continue to next component, next AL section, or start of next cycle
Abort AssemblyLine (AL On Error will be executed)
Available temporary script variables
work
conn
error
*
Hook Flow rev. 6.1
20060519

Appendix C. Server API

Overview

The IBM Tivoli Directory Integrator 6.1 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 and EventHandlers
- Cycle manually through AssemblyLines
- Register for and receive notifications for Server events
- Register for and receive AssemblyLines and EventHandlers 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 or EventHandler hooks and also using the API from new components (Connectors, EventHandlers) 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 emitted by the Server API internal engine are also available as JMX notifications.

© Copyright IBM Corp. 2002, 2006

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:

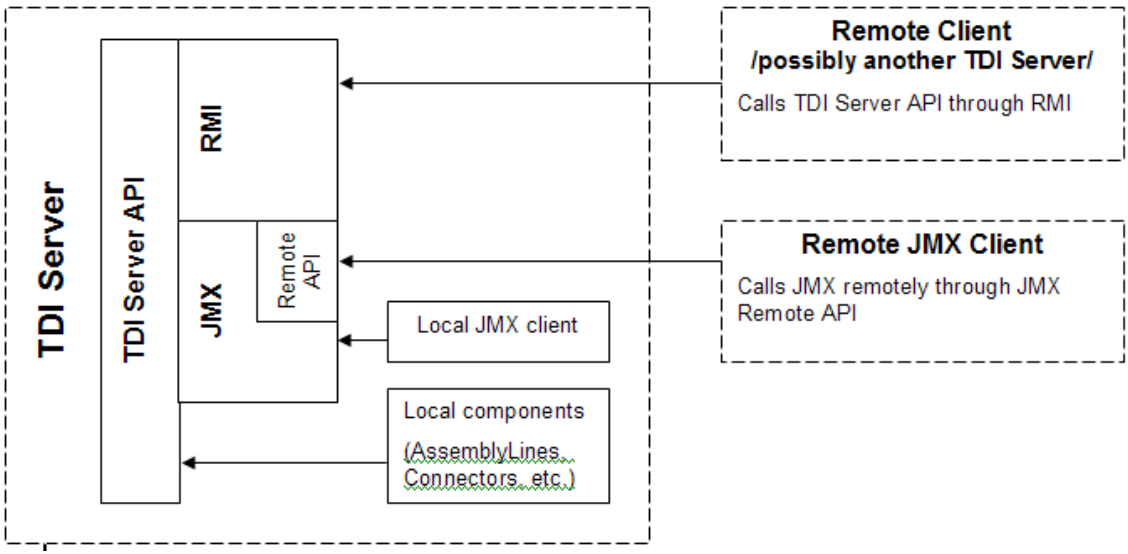


Figure 4.

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 390, 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 and EventHandlers running in the Config Instance as well as calls for starting new AssemblyLines and EventHandlers. 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`. 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.

Configuring the Server API

Configuring the Server API on the Server side includes specifying the relevant system properties in global.properties (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 "Security and TDI chapter, section Server API Access Security" in the *IBM Tivoli Directory Integrator 6.1: Administrator Guide* for information on how to setup the Server API.

Setting up the User Registry

Refer to the "Security and TDI" chapter in the *IBM Tivoli Directory Integrator 6.1: Administrator Guide* for information and examples of how to setup the User Registry, assign user roles and encrypt/decrypt the User Registry file.

Remote client configuration

This section describes the necessary for a remote client that will use the remote Server API.

Prerequisites:

Java 1.4.2 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.jar
   - jars/common/miconfig.jar

- jars/common/miserver.jar
- jars/common/mmconfig.jar
- jars/3rdparty/IBM/icu4j_3_4_1.jar
- jars/3rdparty/IBM/ITLMToolkit.jar

Users can copy these jar files from the TDI 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.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 TDI Server public certificate.
- **api.truststore.pass** – specifies the password for the keystore file specified by *api.truststore*.

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.truststore=C:\TDI\serverapi\testadmin.jks
    -Dapi.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 TDI. 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_por*t 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 or EventHandler is running in the context of a Config Instance. Through a Config Instance you can query the configuration of AssemblyLines, EventHandlers, Connectors, Parsers, Functional Components, start AssemblyLines and EventHandlers, get access to running AssemblyLines and EventHandlers and query their log files.

Getting access to running Config Instances

You can obtain access to all Config Instances running on the TDI 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 and EventHandlers or stop the Config Instance on the Server when you no longer need it.

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, i.e. 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.

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)
```

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).

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();
```

Working with EventHandlers

Everything stated in section "Working with AssemblyLines" about AssemblyLines is valid for EventHandlers as well. You can work with EventHandlers in exactly the same manner using the corresponding EventHandler classes, interfaces and methods. Please consult the JavaDocs for the signatures of the EventHandler related classes, interfaces and methods.

Note: The concept of EventHandlers is deprecated in TDI 6.1 and will be removed in a future version. Use AssemblyLines with Connectors in Server Mode instead.

Editing configurations

TDI Configurations folder

A new TDI Server property `api.config.folder` has been added 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.

TDI 6.1 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. TDI 6.1 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.

New Server API event for configuration update

A new 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

A System Queue TDI module is introduced in TDI 6.1. 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 6.1: 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 sub-systems 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.

A SystemQueue MBean has been introduced in the TDI 6.1 Server API JMX layer. This MBean provides JMX access to the SystemQueue. A JMX client can access the newly introduced 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:

- 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/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 TDI: `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

A Tombstone Manager TDI module was introduced in TDI 6.1.

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, AssemblyLine and EventHandler instances. The GUID is a string value that is unique for each instance of a Config Instance, an AssemblyLine or an EventHandler ever created by a particular TDI Server.

GUIDs are defined as the string representation of the Config Instance/AssemblyLine/EventHandler object hashcode concatenated with the string representation of the Config Instance/AssemblyLine/EventHandler start time in milliseconds.

A new method has been added to the Config Instance, AssemblyLine and EventHandler Server API interfaces: `String getGlobalUniqueID ();`

A new field GlobalUniqueID has been added to AssemblyLine, EventHandler 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 Assembly Line can be retrieved this way (the example Assembly Line is named “myline” and the ID of the configuration is “C__TDI_myconfig.xml”):

```
Tombstone[] alTombstones = tombstoneManager.getAssemblyLineTombstones("AssemblyLines/myline",          "C_
```

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[] getEventHandlerTombstones (String aEventHandlerName, String aConfigID)`

  Returns all available tombstones for the specified EventHandler.
- `Tombstone[] getEventHandlerTombstones (String aEventHandlerName, String aConfigID, java.util.Date aStartTime, java.util.Date aEndTime)`

  Returns all available tombstones for the specified EventHandler 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 new 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 deleteEHTombstones (String aEventHandlerName, String aConfigID)`

  Deletes all tombstones for specified EventHandler. Returns the number of deleted tombstone records.
- `int deleteEHTombstones (String aEventHandlerName, String aConfigID, int aDays)`

  Deletes all tombstones for the specified EventHandler that are older than the specified number of days. Returns the number of deleted tombstone records.
- `int keepMostRecentEHTombstones (String aEventHandlerName, String aConfigID, int aMostResentToKeep)`

  After this method is executed only the *aMostRecentToKeep* most recent tombstone records for the specified EventHandler 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 and EventHandler 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, AssemblyLines and EventHandlers. 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.eh.start** – EventHandler started
  - **di.eh.stop** – EventHandler 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, for EventHandlers this is the EventHandler 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, AssemblyLine or EventHandler. 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 397 describes how listeners can be used to get AssemblyLine (or EventHandler) log messages in real time as they are produced.

The Server API provides another mechanism for direct access to log files produced by AssemblyLines or EventHandlers. This mechanism only provides access to the log files generated by the AssemblyLine or EventHandler SystemLog logger.

You don't need a reference to an AssemblyLine or EventHandler Server API object to get to the log file. Also you can access old logs of AssemblyLines/EventHandlers 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 394 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 same methods are available for EventHandler log files. Consult the JavaDoc of the `com.ibm.di.api.remote.SystemLog` or `com.ibm.di.api.local.SystemLog` interfaces for the signatures and description of the EventHandler methods.

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 and EventHandlers) 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 or EventHandler. You can specify that you want to delete all log files except the 5 most recent logs for all AssemblyLines and EventHandlers:

```
systemLog.cleanAllOldLogs(5);
```

You can also delete the log files for AssemblyLines only or for EventHandlers only or for a specific AssemblyLine or EventHandler. The same two criteria are available: date and number of log files but in addition you can specify the name of an AssemblyLine or EventHandler or use calls that operate on all AssemblyLines or all EventHandlers only. 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, EventHandlers 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,Functional Components and EventHandlers.

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 DIExcept
```

and

```
public Object invokeCustom(String aCustomClassName, String aMethodName, Object[] aParamsValue, String[] a
```

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 6.1: 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, AssemblyLine or EventHandler is wrapped in an MBean. When the Config Instance, AssemblyLine or EventHandler is started the MBean is created automatically and it is automatically removed when the Config Instance, AssemblyLine or EventHandler 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 403.)

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.eh.start – EventHandler started

- di.eh.stop – EventHandler stopped
- di.ci.file.updated – Configration file modified
- di.server.stop – TDI Server shutdown

JMX Example - TDI 6.1 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/solution.properties file:

```
## 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.truststore=testserver.jks
{protect}-api.truststore.pass={encr}L79kdqak1afKdAyuCZBMi1GqY/DPfD1Ipo020CVAGx/OROE2JBUTgZxLjqADXSZJgM3

## 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 ac
## 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.

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 Auth
CTGDKD111I JMX Remote Server Connector started at: service:jmx:rmi://localhost/jndi/rmi://localhost:109
```

MC4J side

1. Download and install MC4J from http://sourceforge.net/projects/mc4j/.
2. Start the **Connect to server ...** wizard

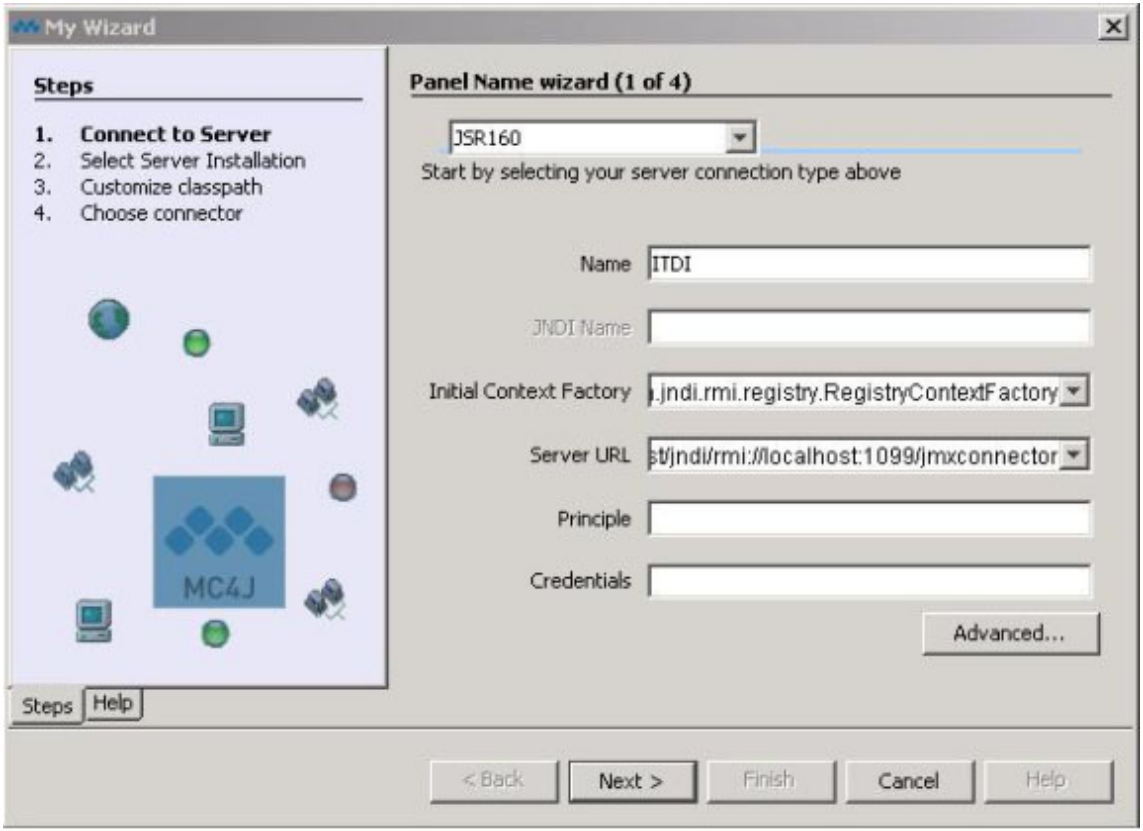


Figure 5.

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**.

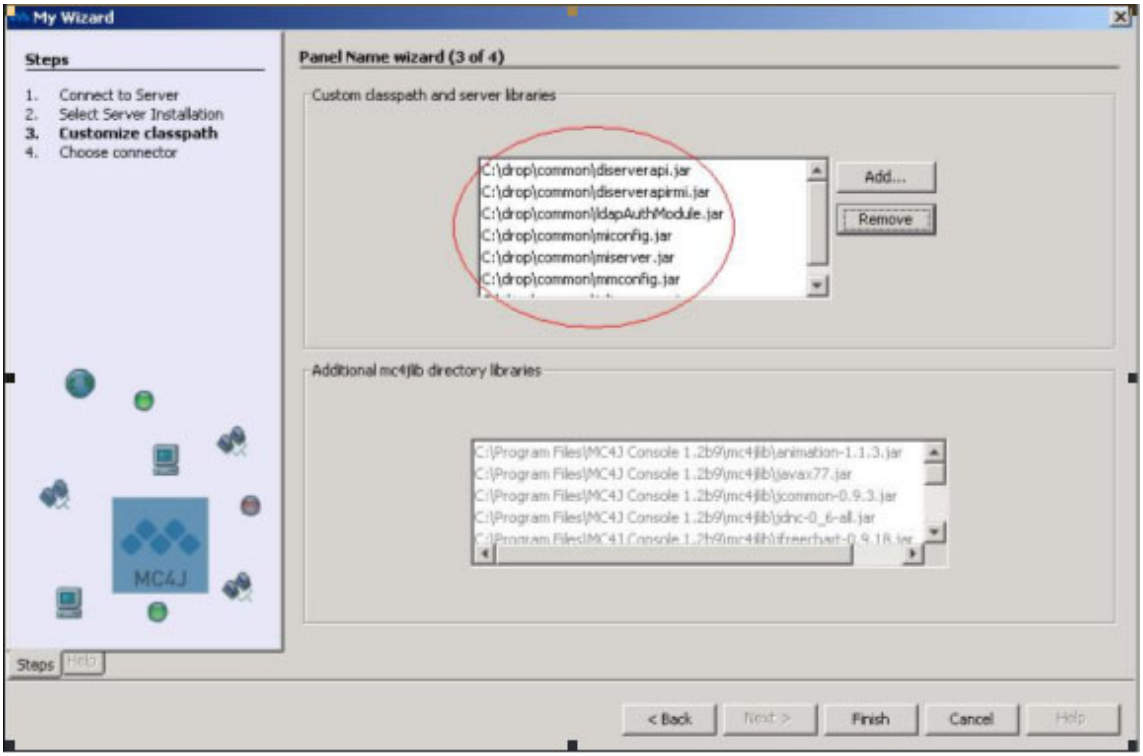


Figure 6.

6. In the **Custom classpath and server libraries** list, add all JAR files from the <*TDI_HOME*>\jars\common folder.
7. Add these three jars as well:
   - <*TDI_home*>\jars\3rdparty\others\log4j-1.2.8.jar
   - <*TDI_home*>\jars\3rdparty\IBM\icu4j_3_4_1.jar
   - <*TDI_home*>\jars\3rdparty\IBM\ITLMToolkit.jar
   - Select **Finish**.

Now MC4J is connected to the TDI server.

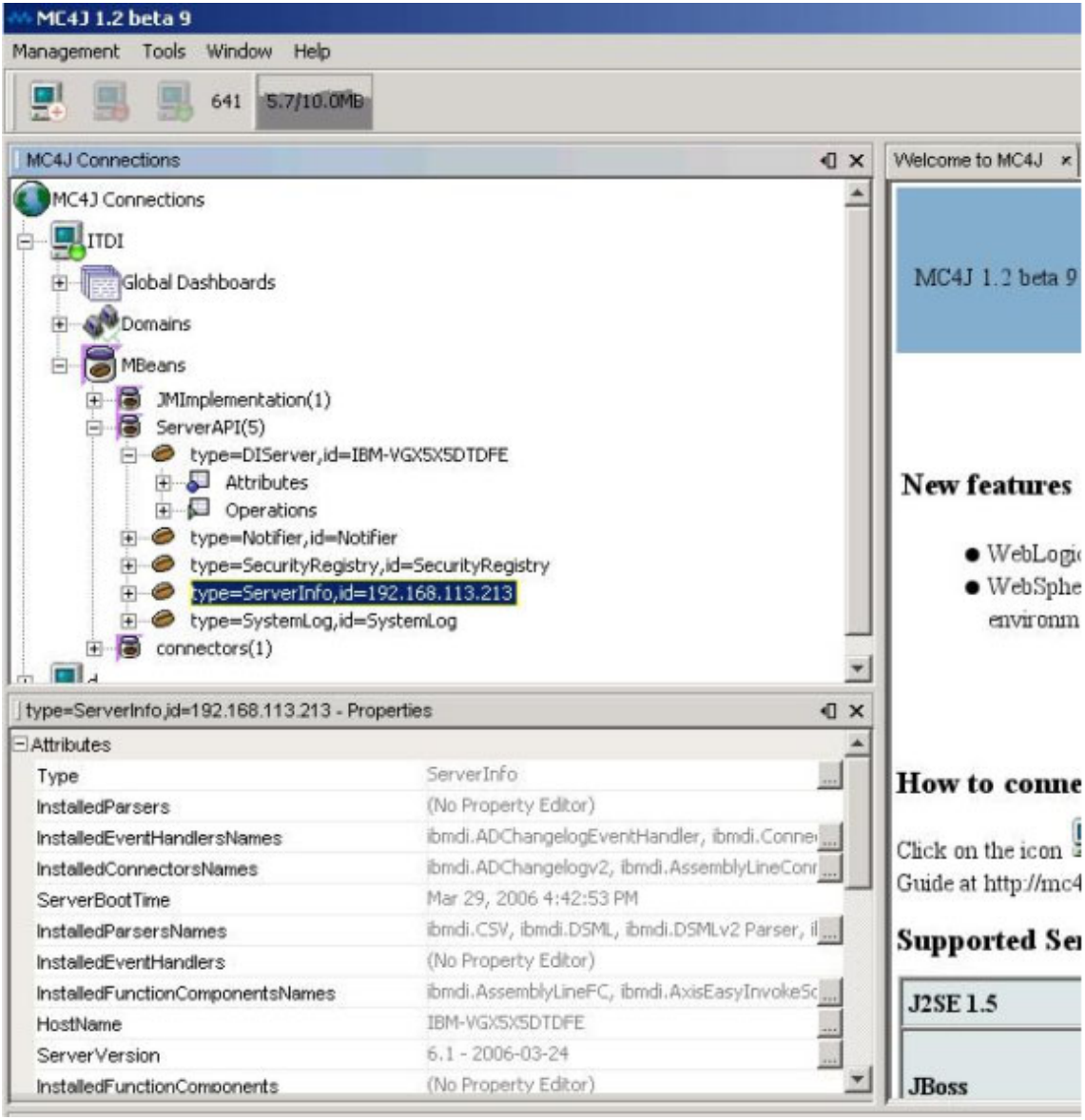

MC4J 1.2 beta 9
Management
Tools
Window
Help
641
5.7/10.0MB
MC4J Connections
MC4J Connections
ITDI
Global Dashboards
Domains
MBeans
JMImplementation(1)
ServerAPI(5)
type=DIServer,id=IBM-VGX5X5DTDFE
Attributes
Operations
type=Notifier,id=Notifier
type=SecurityRegistry,id=SecurityRegistry
type=ServerInfo,id=192.168.113.213
type=SystemLog,id=SystemLog
connectors(1)
type=ServerInfo,id=192.168.113.213 - Properties
Attributes
Type
ServerInfo
InstalledParsers
(No Property Editor)
InstalledEventHandlersNames
InstalledConnectorsNames
ServerBootTime
Mar 29, 2006 4:42:53 PM
InstalledParsersNames
InstalledEventHandlers
(No Property Editor)
InstalledFunctionComponentsNames
HostName
IBM-VGX5X5DTDFE
ServerVersion
6.1 - 2006-03-24
InstalledFunctionComponents
(No Property Editor)
Welcome to MC4J
MC4J 1.2 beta 9
New features
How to conne
Supported Se
J2SE 1.5
JBoss

Figure 7.

Backward compatibility

Scenarios overview

While upgrading your TDI 6.0 installation to TDI 6.1, you may find yourself in one of the following scenarios:

Table 14.

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 TDI 6.1 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 TDI 6.1 servers" on page 424" section	OK

Guidelines for porting a TDI 6.0 Server API client to use a TDI 6.1 server

When is porting required: Probably the most significant change in the TDI 6.1 Server API is the way configurations are edited. For more information on editing configurations in TDI 6.1 see the "Using the Server API -> Editing Configurations" section.

Server API changes in TDI 6.1 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 17 on page 428 - Changed Methods".
- If a TDI 6.0 client uses some of the method calls listed in "Table 18 on page 429 - Deprecated methods" it needs to be reworked to use the new TDI 6.1 methods instead.

Another reason to rework a TDI 6.0 client is to benefit from the functionality introduced in TDI 6.1:

- There are several interfaces introduced in TDI 6.1 described in "Table 15 on page 426 – 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 16 on page 426 - 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 424" 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 19 on page 429 - 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 TDI 6.1 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 TDI 6.1 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 TDI 6.1 features against a TDI 6.1 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 425" 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 TDI 6.1 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 TDI 6.1 (as they are described in "Table 16 on page 426 - New Methods"):

- The methods introduced in TDI 6.1 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 18 on page 429 - Deprecated methods" can only be used with a TDI 6.0 server. If a deprecated method is invoked on a TDI 6.1 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 TDI 6.1. Thus these classes are different in TDI 6.0 and in TDI 6.1. 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 TDI 6.1 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 20 on page 429 – serialVersionUID for serializable classes". Since the TDI 6.1 serializable classes are compatible with the TDI 6.0 serializable classes the serial version UIDs of these classes have not changed.

"Table 19 on page 429 - New Serializable classes/interfaces" lists classes and interfaces introduced in TDI 6.1. 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_

Config Editing: Config editing in TDI 6.1 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 TDI 6.1 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 TDI 6.1 servers. Sometimes, however, TDI 6.0- or TDI 6.1-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 TDI 6.1

Table 15. 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 16. 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.
EventHandler	
String getGlobalUniqueID ()	Returns EventHandler GUID. The GUID is a string value that is unique for each component ever created by a particular TDI Server.
Session	

Table 16. 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.

Table 16. New Methods (continued)

Name	Description
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.
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 17. Changed Methods

Name	Description
ConfigInstance	

Table 17. Changed Methods (continued)

Name	Description
void setConfiguration (MetamergeConfig aConfiguration)	In TDI 6.1 this method can be invoked only if a particular client has already checked out same config with temporary config instance.

Table 18. Deprecated methods (these are methods which are not to be used against a TDI 6.1 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 19. New Serializable classes/interfaces

Name	Description
com.ibm.di.api.Tombstone	5178569311755396746L
com.ibm.di.api.CIEvent	5178569311755396746L
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	55949197177690302911
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	-3361471837888677277L
com.ibm.di.config.interfaces.TDIPropertyStore	198251115520372634L
com.ibm.di.config.interfaces.TombstonesConfig	-3260102686391332434L

Table 20. serialVersionUID for serializable classes

Name	Status	serialVersionUID
com.ibm.di.api.ALEvent	backward compatible	5631772256973692972L

Table 20. serialVersionUID for serializable classes (continued)

Name	Status	serialVersionUID
com.ibm.di.config.base.ALMappingConfigImpl	backward compatible	2712493657450710788L
com.ibm.di.server.ALState	backward compatible	6699383122608684 91L
com.ibm.di.config.base.AssemblyLineConfigImpl	backward compatible	2715909691453046036L
com.ibm.di.entry.Attribute	backward compatible	66758817449018603 29L
com.ibm.di.config.base.AttributeMapConfigImpl	backward compatible	-2619015538178665684L
com.ibm.di.entry.AttributeValue	backward compatible	100100L
com.ibm.di.config.base.BaseConfigurationImpl	known issue – see the "Known issues" on page 432 section	-7316979979253125005L
com.ibm.di.config.base.BranchConditionImpl	backward compatible	-40917732335838179 12L
com.ibm.di.config.base.BranchingConfigImpl	backward compatible	-10135888843811339 44L
com.ibm.di.config.base.CallConfigImpl	backward compatible	-46974584978353290 96L
com.ibm.di.config.base.CallParamConfigImpl	backward compatible	57880211547147417 67L
com.ibm.di.config.base.CheckpointConfigImpl	backward compatible	-83423698815234684 83L
com.ibm.di.config.base.ConfigCache	backward compatible	-33112557315041744 16L
com.ibm.di.config.base.ConfigStatistics	backward compatible	-12716454573849112 49L
com.ibm.di.config.base.ConnectorConfigImpl	backward compatible	40933764562122300 00L
com.ibm.di.config.base.ConnectorSchemaConfigImpl	backward compatible	9301612918007529 10L
com.ibm.di.config.base.ConnectorSchemaItemConfigImpl	backward compatible	-16655981947572957 69L
com.ibm.di.config.base.ContainerConfigImpl	backward compatible	-41340044095926940 52L
com.ibm.di.config.base.DeltaConfigImpl	backward compatible	-72501284845880240 17L
com.ibm.di.api.DIEvent	backward compatible	-86645334774524912 19L
com.ibm.di.api.EHEvent	backward compatible	-16902704612639181 59L
com.ibm.di.entry.Entry	backward compatible	-59614245293786257 29L
com.ibm.di.config.base.EventHandlerActionConfigImpl	backward compatible	-17984970783409458 64L
com.ibm.di.config.base.EventHandlerActionItemConfigImpl	backward compatible	76697455048979068 57L
com.ibm.di.config.base.EventHandlerConditionConfigImpl	backward compatible	14272054080433125 36L
com.ibm.di.config.base.EventHandlerConfigImpl	backward compatible	-42740350787261486 43L
com.ibm.di.config.interfaces.ExternalPropertiesDelegator	known issue – see the "Known issues" on page 432 section	77251874257313816 60L
com.ibm.di.config.base.ExternalPropertiesImpl	backward compatible	-58376587585253002 21L
com.ibm.di.config.base.FormConfigImpl	backward compatible	-87613496958057050 52L

Table 20. serialVersionUID for serializable classes (continued)

Name	Status	serialVersionUID
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	-916088308989377612L
com.ibm.di.config.interfaces.InheritanceLoopException	backward compatible	-5977834080357995975L
com.ibm.di.config.base.InheritConfigImpl	backward compatible	90155321639831994 87L
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	-92068565361720118 21L
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	6299750464788808971L
com.ibm.di.config.base.LoopConfigImpl	backward compatible	-817454107451048 1418L
com.ibm.di.config.base.MetamergeConfigImpl	backward compatible	-336369533068596 7904L
com.ibm.di.config.xml.MetamergeConfigXML	backward compatible	-44031697115790297 65L
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	-2620929677558833640L
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	5168801947811376566L
com.ibm.di.config.base.ScriptConfigImpl	backward compatible	-77476862425517938 90L
com.ibm.di.api.remote.impl.rmi.SSLRMIClientSocketFactory	backward compatible	5083017546031420384L
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 TDI 6.1 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. Implementing your own Components

This chapter is intended for developers that are tasked with creating new Connectors or Function Components for IBM Tivoli Directory Integrator (TDI). They should have a firm understanding of TDI 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.

TDI currently also supports a component type called an EventHandler; however, this type of component is being phased out and replaced by Connectors in Server mode.

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 TDI classes cited in this chapter are located in the `"/docs/api"` folder of your TDI installation. You can view this documentation by selecting **Help>Low Level API** from within the Config Editor.

Developing a Connector

Implementing the Connector's Java source code

All TDI 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 TDI. Usually the Connectors you write will not require all the options provided by TDI 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:

© Copyright IBM Corp. 2002, 2006

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, 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 (i.e. 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, i.e. 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.

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 Vector 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.)
syntax	The syntax (like **VARCHAR** or **TIMESTAMP**) or expected value type of this attribute.
size	If specified, this will give the user a hint as to how long the field may be.

Specified by: *querySchema* in ConnectorInterface

Parameters: *source* - The object on which to discover schema. 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.

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 TDI installation there will most likely be loader problems during execution.

Implementing the Connector's GUI configuration form

Introduction

When you create a custom TDI 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 *idi.inf*. The syntax and contents of this file is described in this document. Localized versions of this file are provided by inserting the locale identifier in the file name (e.g. idi_en.inf, idi_fr.inf etc).

The first part of this section explains the format of this file and also shows the minimum requirements for a component inf-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 TDI 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.

idi.inf file format

The ".inf" files are created in the original (i.e. IDI 4.6) configuration file format (that is, not XML). The format is a very simple key/value pair file with hashtables and array constructs. These are organized into a tree structure (all lines are trimmed before interpretation) that is used internally by the MetamergeConfig drivers (MetamergeConfigImpl).

The file syntax can be summarized by the following:

- All sections must start with a main section definition and end with a main section terminator

  `[connectors connectorName]` Connector main section

  `[parsers parserName]` Parser main section

  `[functions functionName]` Function main section

  `[end]` Main section terminator
- If the line ends with an opening curly brace it denotes a named Hashtable
- If the line ends with an opening bracket it denotes an array of values
- Otherwise, it is expected to be a *keyword:value* pair. Keyword/value pairs cannot span lines (e.g. there is no line-continuation character) so you have to add a "\n" if you need a newline in the text.

An example:

```
[connectors CustomConnector]
  connectorConfig {
   connectorType:pub.test.CustomConnector
   options [
    First Value
    Second Value
    Third Value
   ]
   subsection {
    anotherKey:anotherValue
   }
   paramWithNewlines:line 1\nline 2\n line3
  }
[end]
```

The above constructs a Hashtable with one key ("connectorConfig") that contains three named items (mode, options and subsection). This hashtable is added as a connector named CustomConnector to the current configuration (MetamergeConfig object). The specific

parameters for your component appear inside the connectorConfig/functionConfig/ parserConfig sections. In the above example options, subsection and *paramWithNewlines* are all component specific parameters.

Basic Component Definitions: When you first create your inf-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 21.

Component Type	Minimum Section Contents

Connector	`[connectors CustomConnector]`
	` connectorConfig {`
	`  connectorType:pub.test.CustomConnector`
	` }`
	`}`
	`[end]`

Parser	`[parsertypes CustomParser]`
	` parserConfig {`
	`  class:com.ibm.di.parser.FixedRecordParser`
	` }`
	`[end]`

Function	`[functions ibmdi.ComplexTypesGenerator]`
	` functionConfig {`
	`  javaclass:com.ibm.di.fc.webservice.ComplexTypesGenerator`
	` }`
	`[end]`

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 *FormUI* 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 TDI Loader that runs through its configured jar directories looking for *.jar/*.zip files that contain an "idi.inf" file at its root level. All these files are concatenated into a single configuration file known as the system templates (the system namespace in configuration terms).

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 (defined in standard_templates.cfg / miadmin.jar), most components need specific user interfaces to let the user define its behaviour.

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 (e.g. 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 (pub.test.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 (e.g. 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.

```
[form pub.test.CustomConnector]

 title:This is the title/heading that appears at the top of the form

 formevents:function firstParameter_changed() { form.alert("First param modified"); }
 parameterlist [
  firstParameter
  secondParameter
  $GLOBAL.debug
 ]
 parameters {
  firstParameter {
   label:First Param Label
```

```
  }
  secondParameter {
   label:Second Param Label
  }
 }
[end]
```

Form - top level keywords: At the top level of the form definition you define the overall characteristics of the form.

Table 22.

Keyword	Description
title	The heading text for the form.
width/height	If the form is presented in its own frame (e.g. modal dialog etc) these fields define the size of the frame.
parameterlist	This array defines which parameters are included in the form and order in which they appear. Parameters starting with ″$GLOBAL.″ refers to commonly used parameter settings, which is defined in the ″standard_forms.cfg″ resource (miadmin.jar). For each parameter listed here there must be a corresponding subsection in the parameters section.
parameters	This section contains the specific configurations for each parameter. See parameter definitions next.
formevents	This keyword lets you define javascript functions that are called when parameter values change. The function name is constructed using the parameter name plus a ″_changed″ suffix. So to intercept changes to ″firstParameter″ you would add a function called ″firstParameter_changed()″. In the function you have access to the ″form″ object which is the instance of the com.ibm.di.admin.ui.FormUI class shown in the configuration tab of the component.
tablist	This parameter lets you create a multi-form form. The value of the tablist specifies the name of each form which will be created and added as a separate pane to the main form. Each sub-form shares the same configuration object. When this parameter is present only the sub-forms are created. That means that any parameterlist/parameters sections are ignored for this form definition.
<form>.title <form>.tooltip	When *tablist* is specified these parameters define the title and tool tip for each sub-form. If tablist contains the form “A” you can define “A.title” and “A.tooltip” to customize the tab header in the main form.

Parameter Definitions: Each parameter has its own section within the parameters section. In this section you define the characteristics for each parameter.

Table 23.

Keyword	Description
label	The label appearing in the left column of the form (e.g. 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 idi.inf 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.
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.
values	Array of values used to populate dropdown lists. The values can be static or dynamic. Dynamic values are values that the FormUI will gather from configuration objects or from java objects passed to the FormUI when it was instantiated (this is typically config editor specific and not used much by 3rd party devs). See section below for possible macros for dynamic value lists.
localizedValues	Array of *display* values. This field is only used when the values parameter is provided. Instead of showing the user the text from the values parameter the corresponding value in this array is shown instead (index based). This is often used to provide localized values and/or human readable text for numeric/technical configuration values.
readonly	If present and set to “true” the input field is not editable.
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 (e.g. 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.
reflectClass	Optional – in case there are several versions of the reflect method this parameter specifies the conversion class (e.g. java.lang.Boolean) to be used when setting the parameter value (e.g. setValue(boolean) vs setValue(string).
minValue maxValue	Used when the parameter syntax is Number. The fields specify the minimum and maximum numeric value.

Dynamic Values: The *values* array can contain static and dynamic values. The dynamic values are expanded and added to the array at runtime to populate the dropdown list.

Table 24.

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
@USER.PROPERTIES@	Adds all known property values to the array
@CONFIG:param	Retrieves the value for param from the current configuration object and adds it to the array.

Syntax: The syntax for a parameter can be any of the following.

Table 25.

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.
Fontlist Fontedit	Creates a dropdown list of available fonts
Color	Creates a Color selection dialog
List	Creates a JList showing all values with the selection as the current value
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 string w/readonly=true)

Table 25. (continued)

Value	Description
Number	Creates a single-line text input field with validation. Validation includes verifying the input is a valid numeric string in the range defined by *minValue* and *maxValue*
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 (e.g. 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
	
	The class is instantiated by FormUI at runtime and should be an AWT or Swing subclass (something that can be added to a JPanel). Also, it must have a constructor as shown in this example:
	`package pub.test;`
	
	`import com.ibm.di.admin.ui.FormUI;`
	`import com.ibm.di.config.interfaces.BaseConfiguration;`
	
	`public class CustomUI extends JPanel {`
	`/*`
	`* form – the FormUI object`
	`* config – the config object being modified`
	`* paramname – the parameter of config being edited`
	`*/`
	`public CustomUI(FormUI form, BaseConfiguration config, String paramname)`
	`}`
	
	`}`

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.di.admin.ui.FormUI instance managing this form
- **config** A handle to the configuration objects this form operates on (e.g. the connection configuration for a connector, parser config for parser etc)
- **button** The JButton that triggered the script to be invoked
- **util** An instance of the com.ibm.di.admin.ui.Util class
- **system** An instance of the com.ibm.di.function.UserFunctions class

Multi-Form Definition: Your component's inf-file can define more than one form. Often, it is advisable to break a form into several parts (like common, advanced etc) instead of having one form with too many input fields. To accomplish this you basically create a separate form definition for each part and bring them together in the main form. You are free to choose the name of your sub-forms but to avoid name collisions with other components you should use your main form name as base. Here is an example of a multi-form with two sub-forms.

```
[form pub.test.CustomConnector]
 tablist:pub.test.CustomConnector.common, pub.test.CustomConnector.advanced

 pub.test.CustomConnector.common.title:Common Parameters
  pub.test.CustomConnector.common.tooltip:This tab contains the most commonly used params

 pub.test.CustomConnector.advanced.title:Advanced Parameters pub.test.CustomConnector.advanced.tooltip:Th
[end]

[form pub.test.CustomConnector.common]
 (form definition)
[end]

[form pub.test.CustomConnector.advanced]
 (form definition)
[end]
```

Examples

Look at TDI's components in the configuration editor to find an example you find suitable. Use a zip/jar tool (e.g. winzip, unzip) and extract the "idi.inf" file from the component's jar file (<tdi_installdir>/jars/components subdirectory).

Also, the *"examples/connector_java"* folder of this package contains the `"idi.inf"` file of the Directory Connector.

Packaging and deploying the Connector

Now that we have the Connector source code compiled and supplied the `"idi.inf"` 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 `"idi.inf"` file, in the root of the jar file. (You might also include `"idi.inf"` files for different languages naming them in the standard Java internationalization schema – for example `"idi_de.inf"` for German, `"idi_fr.inf"` for French, etc.)

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 absolutely the same 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). 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. 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, i.e. 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.

public java.awt.Component getUI()

The *"getUI(...)"* method allows you to build custom graphical user interface for the configuration form of your FC instead of using the standard configuration forms described in `"idi.inf"` files.

This however is an option. In most cases the standard GUI provided by forms that you can describe in `"idi.inf"` will be sufficient. It is up to you to decide how to implement it, what visual controls to include and how they map to the FC's parameters

Building the Function Component source code

When building the source code of your Connector, include in your `CLASSPATH` the jar files from the `"jars"` 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 TDI installation there will most likely be loader problems during execution.

Implementing the Function Component GUI configuration form

You have two options for implementing the FC GUI – use the standard syntax of `"idi.inf"` file or code your own GUI by implementing the *"getUI(...)"* method.

If you choose to use the standard `"idi.inf"` mechanism, you describe the FC configuration form by using the same syntax as used for Connectors.

Remember that even if you code a custom GUI by implementing the *"getUI(...)"* method, you still need the `"idi.inf"` file in the jar file of your FC. The presence of this file tells TDI that this component is an FC and must be loaded.

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. `"idi.inf"` file placed in the root of the jar file (and optionally `"idi_??.inf"` 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.

See also

Appendix C, "Server API," on page 387

Appendix E. 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, 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.

© Copyright IBM Corp. 2002, 2006

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.

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
OS/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.

Printed in USA

SC32-2566-00