Oracle® GoldenGate for Java

Administrator's Guide

11*g* Release 1 (11.1.1)

E17814-01

August 2010

ORACLE

Oracle GoldenGate for Java Administrator's Guide 11*g* Release 1 (11.1.1)

E17814-01

Copyright © 1995, 2010 Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Chapter 1 Introduction 10

Oracle GoldenGate 10

Adapter integration options. 10

Capturing transactions to a trail 10

Applying transactions from a trail 11

Oracle GoldenGate VAM message capture 11

Overview of capture from JMS messaging 11

Message capture configuration options 12

Oracle GoldenGate Java user exit 12

Delivery to Java messaging or custom applications 12

Delivery configuration options 13

Oracle GoldenGate documentation. 14

Chapter 2 Installing Oracle GoldenGate for Java 15

Preparing for installation 15

Installing Java 15

Java on Windows 15

Java on Linux/UNIX 16

Installing Oracle GoldenGate core software 16

Installing Oracle GoldenGate for Java. 17

Populating the Oracle GoldenGate directory 17

Installing the Java application. 18

Chapter 3 VAM: Configuring Message Capture 19

Configuring the VAM Extract. 19

Adding the Extract. 19

Configuring the Extract parameters 19

Configuring the message capture. 20

Connecting and retrieving the messages. 20

Connecting to JMS. 20

Retrieving messages. 21

Completing the transaction 21

Chapter 4 VAM: Parsing the message 22

Parsing overview. 22

Parser types 22

Source and target data definitions ... 22
Required Data ... 23
Optional data ... 24
Fixed width parsing ... 25
Header ... 25
Header and record data type translation ... 27
Key identification ... 27
Delimited parsing ... 27
Metadata columns ... 28
Parsing properties ... 28
Parsing steps ... 29
XML parsing ... 29
Styles of XML ... 29
XML parsing rules ... 30
XPath expressions ... 31
Other value expressions ... 32
Transaction rules ... 33
Operation rules ... 33
Column rules ... 34
Overall rules example ... 35
Source definitions generation utility ... 36
Chapter 5 VAM: Message Capture Properties ... 37
Logging and connection properties ... 37
Logging properties ... 37
log.logname ... 37
log.level ... 38
log.tostdout ... 38
log.tofile ... 38
log.modules, log.level.{module} ... 38
JMS connection properties ... 38
jvm.boot options ... 39
jms.report.output ... 39
jms.report.time ... 39
jms.report.records ... 39
jms.id ... 40
jms.destination ... 40
jms.connectionFactory ... 40
jms.user, jms.password ... 40

JNDI properties 41
Parser properties 41
Setting the type of parser 41
parser.type 41
Fixed parser properties 42
fixed.schema 42
fixed.sourcedefs 42
fixed.copybook 42
fixed.header 42
fixed.seqid 42
fixed.timestamp 43
fixed.timestamp.format 43
fixed.txid 43
fixed.txowner 44
fixed.txname 44
fixed.optype 44
fixed.optype.insertval 44
fixed.optype.updateval 44
fixed.optype.deleteval 45
fixed.table 45
fixed.schema 45
fixed.txind 45
fixed.txind.beginval 46
fixed.txind.middleval 46
fixed.txind.endval 46
fixed.txind.wholeval 46
Delimited Parser Properties 46
delim.sourcedefs 46
delim.header 47
delim.seqid 47
delim.timestamp 47
delim.timestamp.format 47
delim.txid 48
delim.txowner 48
delim.txname 48
delim.optype 48
delim.optype.insertval 49
delim.optype.updateval 49
delim.optype.deleteval 49

delim.schemaandtable ... 49
delim.schema ... 49
delim.table ... 50
delim.txind ... 50
delim.txind.beginval ... 50
delim.txind.middleval ... 50
delim.txind.endval ... 50
delim.txind.wholeval ... 51
delim.fielddelim ... 51
delim.linedelim ... 51
delim.quote ... 51
delim.nullindicator ... 51
delim.fielddelim.escaped ... 51
delim.linedelim.escaped ... 52
delim.quote.escaped ... 52
delim.nullindicator.escaped ... 52
delim.hasbefores ... 52
delim.hasnames ... 53
delim.afterfirst ... 53
delim.isgrouped ... 53
delim.dateformat ... 53
XML parser properties ... 54
xml.sourcedefs ... 54
xml.rules ... 54
{rulename}.type ... 54
{rulename}.match ... 54
{rulename}.subrules ... 55
{txrule}.timestamp ... 55
{txrule}.timestamp.format ... 55
{txrule}.seqid ... 56
{txrule}.txid ... 56
{txrule}.txowner ... 56
{txrule}.txname ... 57
{oprule}.timestamp ... 57
{oprule}.timestamp.format ... 57
{oprule}.seqid ... 57
{oprule}.txid ... 58
{oprule}.txowner ... 58
{oprule}.txname ... 58

{oprule}.schemandtable ... 58
{oprule}.schema ... 59
{oprule}.table ... 59
{oprule}.optype ... 59
{oprule}.optype.insertval ... 59
{oprule}.optype.updateval ... 59
{oprule}.optype.deleteval ... 60
{oprule}.txind ... 60
{oprule}.txind.beginval ... 60
{oprule}.txind.middleval ... 60
{oprule}.txind.endval ... 60
{oprule}.txind.wholeval ... 61
{colrule}.name ... 61
{colrule}.index ... 61
{colrule}.value ... 61
{colrule}.isnull ... 62
{colrule}.ismissing ... 62
{colrule}.before.value ... 62
{colrule}.before.isnull ... 62
{colrule}.before.ismissing ... 62
{colrule}.after.value ... 62
{colrule}.after.isnull ... 63
{colrule}.after.ismissing ... 63

Chapter 6 UE: Configuring for Message Delivery ... 64

Configure the JRE in the user exit properties file ... 64
Configure a data pump to run the user exit ... 64
Configure the Java handlers ... 66

Chapter 7 UE: Running the User Exit ... 67

Starting the application ... 67
Restarting the application at the beginning of a trail ... 67

Chapter 8 UE: Configuring Event Handlers ... 69

Specifying event handlers ... 69
JMS handler ... 70
File handler ... 71
Custom handlers ... 72
Formatting the output ... 72
Reporting ... 72

Chapter 9 UE: Message Delivery Properties . . . 73
User exit properties . . . 73
Logging properties . . . 73
log.logname . . . 73
log.level . . . 74
log.tostdout . . . 74
log.tofile . . . 74
log.modules, log.level.{module} . . . 75
General properties . . . 75
goldengate.userexit.writers . . . 75
goldengate.userexit.chkptprefix . . . 75
JVM boot options . . . 75
javawriter.bootoptions . . . 75
Statistics and reporting . . . 76
javawriter.stats.display . . . 76
javawriter.stats.full . . . 76
javawriter.stats.{time, numrecs} . . . 76
Java application properties . . . 77
Properties for all handlers . . . 77
gg.handlerlist . . . 77
gg.handler.{name}.type . . . 77
Properties for formatted output . . . 78
gg.handler.{name}.format . . . 78
gg.handler.{name}.includeTables . . . 78
gg.handler.{name}.excludeTables . . . 79
gg.handler.{name}.mode, gg.handler .{name}.format.mode . . . 79
Properties for CSV and fixed-format output . . . 79
delim . . . 79
quote . . . 79
metacols . . . 80
missingColumnChar, presentColumnChar, nullColumnChar . . . 80
beginTxChar, middleTxChar, endTxChar, wholeTxChar . . . 80
insertChar, updateChar, deleteChar . . . 80
endOfLine . . . 81
justify . . . 81
includeBefores . . . 81
File writer properties . . . 81
gg.handler.{name}.file . . . 81
gg.handler.{name}.append . . . 81

gg.handler.{name}.rolloverSize 81
JMS handler properties 82
Standard JMS settings 83
gg.handlerlist.nop 84
General properties 84
gg.classpath 84
gg.report.format 84
Chapter 10 UE: Developing Custom Filters, Formatters and Handlers 85
Filtering events 85
Custom formatting 85
Coding a custom formatter in Java 85
Using a Velocity template 87
Coding a custom handler in Java 88
Additional resources 90
Chapter 11 Troubleshooting 92
Error handling 92
Reporting issues 93

CHAPTER 1
Introduction

This guide covers:

- Installing, configuring and running the Oracle GoldenGate for Java
- Using the prebuilt Java Message Service (JMS) and file handlers
- Developing custom filters, formatters or event handlers

Oracle GoldenGate

The core Oracle GoldenGate product:

- Captures transactional changes from a source database. For most databases this is accomplished by reading the database transaction log.
- Sends and queues these changes as a set of database-independent files called the Oracle GoldenGate trail.
- Optionally alters the source data using mapping parameters and functions.
- Applies the transactions in the trail to a target system database

Oracle GoldenGate performs this capture and apply in near real-time across heterogeneous databases and operating systems.

Adapter integration options

The Oracle GoldenGate adapters integrate with installations of the Oracle GoldenGate core product to either 1) read JMS messages and deliver them as an Oracle GoldenGate trail, 2) read an Oracle GoldenGate trail and deliver transactions to a JMS provider or other messaging system or custom application or 3) read an Oracle GoldenGate trail and write transactions to a flat file that can be used by other applications.

Capturing transactions to a trail

The Oracle GoldenGate message capture adapter can be used to read messages from a queue and communicate with an Oracle GoldenGate Extract process to generate a trail containing the processed data.

The message capture adapter is implemented as a Vendor Access Module (VAM) plug-in to a generic Extract process. A set of properties, rules and external files provide messaging connectivity information and define how messages are parsed and mapped to records in the target GoldenGate trail.

Currently this adapter supports capture from JMS text messages.

Applying transactions from a trail

The Oracle GoldenGate delivery adapters can be used to apply transactional changes to targets other than a relational database: for example, ETL tools (DataStage, Ab Initio, Informatica), JMS messaging, or custom APIs. There are a variety of options for integration with Oracle GoldenGate:

- *Flat file integration*: predominantly for ETL, proprietary or legacy applications, Oracle GoldenGate for Flat File can write micro batches to disk to be consumed by tools that expect batch file input. The data is formatted to the specifications of the target application such as delimiter separated values, length delimited values, or binary. Near real-time feeds to these systems are accomplished by decreasing the time window for batch file rollover to minutes or even seconds.
- *Messaging*: transactions or operations can be published as messages (e.g. in XML) to JMS. The JMS provider is configurable; examples include ActiveMQ, JBoss Messaging, TIBCO, WebLogic JMS, WebSphere MQ and others.
- *Java API*: custom event handlers can be written in Java to process the transaction, operation and metadata changes captured by Oracle GoldenGate on the source system. These custom Java handlers can apply these changes to a third-party Java API exposed by the target system.

Oracle GoldenGate VAM message capture

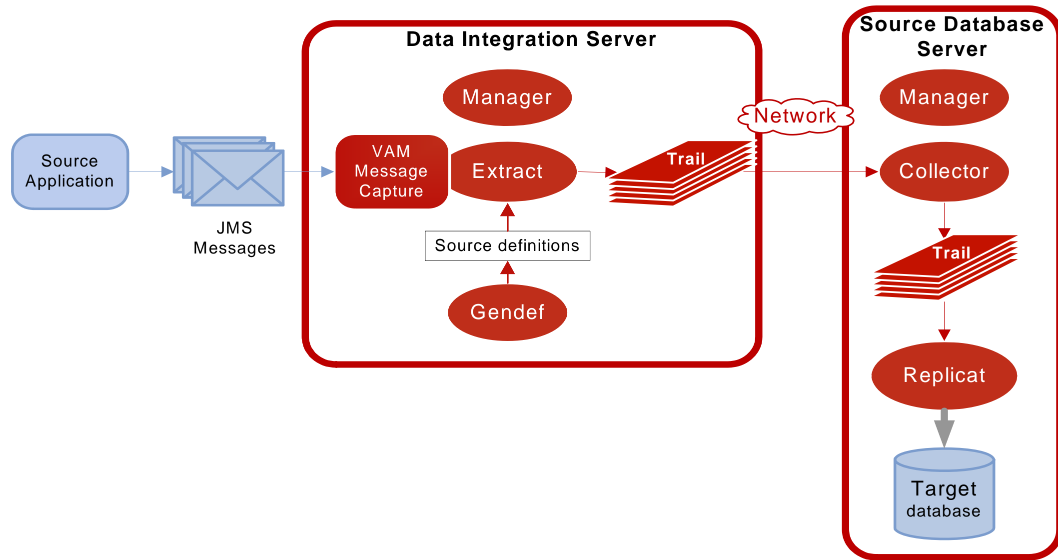
Overview of capture from JMS messaging

The Oracle GoldenGate messaging capture adapter connects to JMS messaging to parse messages and send them through a VAM interface to an Oracle GoldenGate Extract that builds an Oracle GoldenGate trail of message data. This allows JMS messages to be delivered to an Oracle GoldenGate system running for a target database.

Using Oracle GoldenGate JMS message capture requires two components:

- The dynamically linked shared VAM library that is attached to the Oracle GoldenGate Extract process.
- A separate utility, Gendef, that uses the message capture properties file and parser-specific data definitions to create an Oracle GoldenGate source definitions file.

Figure 1 Configuration for JMS message capture



Message capture configuration options

The options for configuring the three parts of message capture are:

- Message connectivity: Values in the property file set connection properties such as the Java classpath for the JMS client, the JMS source destination name, JNDI connection properties, and security information.
- Parsing: Values in the property file set parsing rules for fixed width, comma delimited, or XML messages. This includes settings such as the delimiter to be used, values for the beginning and end of transactions and the date format.
- VAM interface: Parameters that identify the VAM .dll or .so library and a property file are set for the Oracle GoldenGate core Extract process.

Oracle GoldenGate Java user exit

Delivery to Java messaging or custom applications

Through the Oracle GoldenGate Java API, transactional data captured by Oracle GoldenGate can be delivered to targets other than a relational database, such as JMS (Java Message Service), writing files to disk or integrating with a custom application's Java API.

Oracle GoldenGate for Java provides the ability to execute code written in Java from the Oracle GoldenGate Extract process. Using Oracle GoldenGate for Java requires two components:

- A dynamically linked or shared library, implemented in C/C++, integrating as a **user exit** (UE) with the Oracle GoldenGate Extract process through a C API.
- A set of Java libraries (jars), which comprise the Oracle GoldenGate Java API. This Java framework communicates with the user exit through the Java Native Interface (JNI).

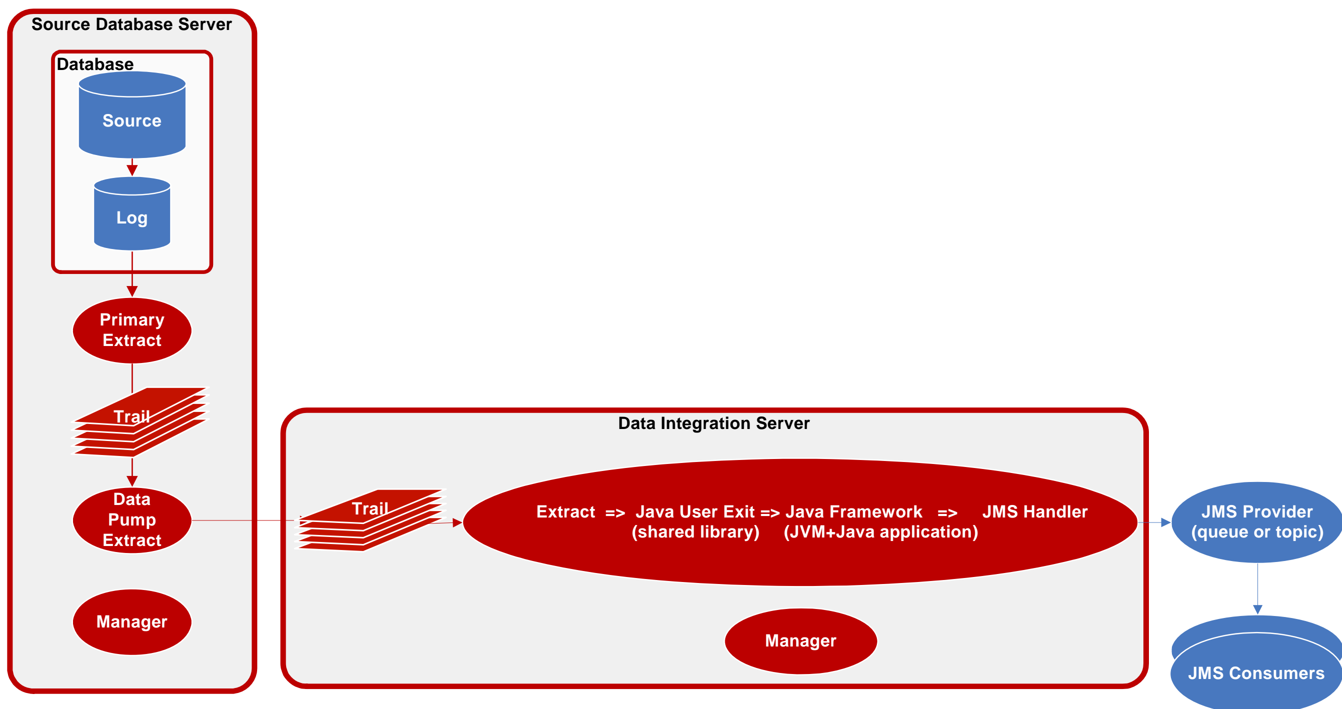


Figure 2 Configuration using the JMS Handler

Delivery configuration options

The dynamically linked library is configurable using a simple properties file. The Java framework is loaded by this user exit and is also initialized by a properties file. Application behavior can be customized by:

- Editing the property files; for example to:
  - Set host names, port numbers, output file names, JMS connection settings;
  - Add/remove targets (such as JMS or files) by listing any number of active handlers to which the transactions should be sent;
  - Turn on/off debug-level logging, etc.
  - Identify which message format should be used.
- Customizing the format of messages sent to JMS or files. Message formats can be custom tailored by:
  - Setting properties for the pre-existing formatters (for fixed-length or field-delimited message formats);
  - Customizing message templates, using the Velocity template macro language;
  - (Optional) Writing custom Java code.
- (Optional) Custom Java code can be written to provide custom handling of transactions and operations, do filtering, or implementing custom message formats.

There are existing implementations (handlers) for sending messages via JMS and for writing out files to disk. There are several predefined message formats for sending the messages (e.g. XML or field-delimited); or custom formats can be implemented using templates. Each handler has documentation that describes its configuration properties; for

example, a filename can be specified for a file writer, and a JMS queue name can be specified for the JMS handler. Some properties apply to more than one handler; for example, the same message format can be used for JMS and files.

Oracle GoldenGate documentation

For information on installing and configuring the core Oracle GoldenGate software for use with the Oracle GoldenGate Flat File or Java adapters, see the Oracle GoldenGate documentation:

- *Installation and Setup guides*: There is one such guide for each database that is supported by Oracle GoldenGate. It contains system requirements, pre-installation and post-installation procedures, installation instructions, and other system-specific information for installing the Oracle GoldenGate replication solution.
- *Oracle GoldenGate Windows and UNIX Administrator's Guide*: Explains how to plan for, configure, and implement the Oracle GoldenGate replication solution on the Windows and UNIX platforms.
- *Oracle GoldenGate Windows and UNIX Reference Guide*: Contains detailed information about Oracle GoldenGate parameters, commands, and functions for the Windows and UNIX platforms.
- *Oracle GoldenGate Windows and UNIX Troubleshooting and Tuning Guide:* Contains suggestions for improving the performance of the Oracle GoldenGate replication solution and provides solutions to common problems.

CHAPTER 2
Installing Oracle GoldenGate for Java

Preparing for installation

Before installing and running Oracle GoldenGate for Java, you must install Java (JDK or JRE) version 1.5 or later.

Installing Java

Java 5 or Java 6 (also known as Java 1.5 and Java 1.6) or later are required. Either the Java Runtime Environment (JRE) or the full Java Development Kit (which includes the JRE) may be used.

- To download Java for Windows, Solaris or Linux, download *either*:
  - The JRE: http://www.java.com/en/download/manual.jsp *or*
  - The Java SDK http://java.sun.com/javase/downloads/index.jsp
- For other platforms, see the OS vendor's support website.

NOTE The Oracle GoldenGate for Java framework has been compiled for Java 5 compatibility, and will therefore work under both Java 5 and Java 6 (or later).

To configure your Java environment for Oracle GoldenGate for Java:

- The PATH environmental variable should be configured to find your Java Runtime and
- The shared (dynamically linked) Java virtual machine (JVM) library must also be found.

On Windows, these environmental variables should be set as system variables; on Linux/UNIX, they should be set globally or for the user running the Oracle GoldenGate processes. Examples of setting these environmental variables for Windows and UNIX/Linux are listed below.

NOTE There may be two versions of the JVM installed when installing Java; one in JAVA_HOME/.../client, and another in JAVA_HOME/.../server. For improved performance, use the server version, if it is available. On Windows, it may be that only the client JVM is there if only the JRE was installed (and not the JDK).

Java on Windows

After Java is installed, configure the PATH to find the 1) JRE and 2) JVM DLL (jvm.dll):

```
set JAVA_HOME=C:\Program Files\Java\jdk1.6.0
set PATH=%JAVA_HOME%\bin;%PATH%
set PATH=%JAVA_HOME%\jre\bin\server;%PATH%
```

Verify the environment settings by opening a command prompt and running: java version:

```
C:\> java -version
java version "1.6.0_05" Java(TM) SE Runtime Environment (build 1.6.0_05-b13)
```

In the example above, the directory `%JAVA_HOME%\jre\bin\server` should contain the file `jvm.dll`

Java on Linux/UNIX

Configure the environment to find 1) the JRE in the PATH, and 2) the JVM shared library, using the appropriate environmental variable for your system. For example, on Linux (and Solaris, etc.), set LD_LIBRARY_PATH to include the directory containing the JVM shared library as follows (for sh/ksh/bash):

```
export JAVA_HOME=/opt/jdk1.6
export PATH=${JAVA_HOME}/bin:${PATH}
export LD_LIBRARY_PATH=${JAVA_HOME}/jre/lib/i386/server:${LD_LIBRARY_PATH}
```

Verify the environment settings by opening a command prompt and running java version:

```
$ java -version
java version "1.6.0_06"
Java(TM) SE Runtime Environment (build 1.6.0_06-b02)
```

In the example above, the directory `$JAVA_HOME/jre/lib/i386/server` should contain the file `libjvm.so`. The actual directory containing the JVM library depends on the OS and if the 32-bit or 64-bit JVM is being used.

Installing Oracle GoldenGate core software

Install Oracle GoldenGate software following the instructions in the appropriate *Oracle GoldenGate Installation and Setup Guide*. The Oracle GoldenGate components may optionally be installed on a separate host from the source database system.

- Extract the zip file (Windows) or tar.gz file (Linux/UNIX) into a directory on your source database system.

  NOTE There cannot be spaces in the path.

- Follow the instructions in the installation guide to create the required subdirectories, create a Manager parameter file and start the Manager process.
- Create a Manager parameter file, specifying a port to listen on, for example:

  ```
  GGSCI> EDIT PARAM MGR
  PORT 7801
  ```

- Start Manager from GGSCI:

  ```
  GGSCI>START MGR
  ```

Installing Oracle GoldenGate for Java

After Oracle GoldenGate and Java are installed, the Oracle GoldenGate adapters can be installed and used. Oracle GoldenGate adapters are available for Windows, Linux and UNIX; but as the shared library is a platform-dependent C library, it must be built specifically for your operating system, hardware architecture, Oracle GoldenGate version and Java runtime version.

Check to see if a build of Oracle GoldenGate for Java is available for your OS, version of Oracle GoldenGate, and version of Java.

The Oracle GoldenGate for Java adapter includes two types of components:

1. Shared libraries (implemented in C/C++) dynamically linked into the Extract process at runtime. This loads the JVM into memory.
2. A set of Java jars for the Java API, including:
   - Existing modules configured via property files (e.g. for JMS integration) and
   - The Java API for Oracle GoldenGate, for implementing custom logic in Java.

Populating the Oracle GoldenGate directory

Oracle GoldenGate for Java is distributed as a zip file containing the platform-dependent shared libraries

- VAM library (libggjava_vam.so or ggjava_vam.dll),
- User exit library (libggjava_ue.so or ggjava_ue.dll)

The Java jars (platform independent) are also distributed as a zip file.

NOTE The platform-independent Java jars and the platform-dependent shared library may have been distributed as a single zip, or distributed separately. When updating the jars or the user exit, they can be updated separately, as long as the versions are compatible.

Extract these zip files into the Oracle GoldenGate installation directory. The shared libraries should be moved to the `dirprm` directory once the GGSCI command CREATE SUBDIRS has been executed. The installation location of the related Java far files can be configured. The following example is a suggested layout.

Table 1 Sample installation directory structure: Oracle GoldenGate, the User Exit and the Java jars

Directory	Explanation

`{gg_install_dir}`
`|-ggsci.exe`
`|-mgr.exe`
`|-extract.exe`
`|-defgen.exe`
`|-gendef.exe`
`|-ggjava_vam.dll`
`|-ggjava_ue.dll`
`| ...`
	Oracle GoldenGate installation directory, containing all Oracle GoldenGate executables. For example: C:/ggs (Windows) or /home/user/ggs (UNIX). Use the GGSCI command line interface to start Extract (as normal) and Extract will in turn start the Java application.
	The user exit and VAM Extract parameter files specify the shared library to load. The JMS message capture shared library is ggjava_vam.dll (Windows) or libggjava_vam.so for UNIX/Linux. The user exit shared library is ggjava_ue.dll (Windows) or libggjava_ue.so (UNIX/Linux).

`|-[dirdat]`
`| |aa000000`
`| |aa000001`
`| ...`
	The VAM Extract parses messages and creates an Oracle GoldenGate trail in the dirdat directory.
	The Extract running the user exit is configured as a data pump, consuming trail data (produced by the primary Extract) in the dirdat directory.

`|-[dirdef]`
`| |-mysrcdefs.def`
`| ...`
	The metadata (column names, data types) for the user exit trail data can come from a source definitions file (e.g. mysrcdefs.def, generated by Defgen) or from the database. For the VAM it can come from a source definitions file generated by Gendef using the properties file.

`|-[dirprm]`
`| |-javaue.prm`
`| |-jmsvam.prm`
`| |-jmsvam.properties`
`| |-javaue.properties`
`| ...`
	The directory for parameter files and property files.
	javaue.properties is a sample user exit properties file. jmsvam.properties is a sample VAM properties file.

`|-[ggjava]`
`|-ggjava.jar`
`|-[resources]`
`|-[config]`
`|`
`|-[classes]`
`|-[lib]`
`|...`
	Installation directory for Java jars (as specified in the user exit properties);
	◆ ggjava.jar – main Java application jar, defines classpath and dependencies
	◆ resources directory – (in classpath) contains all ggjava.jar dependencies:
	resources/classes/* – (in classpath) properties and resources
	resources/lib/*.jar – application jars required by ggjava.jar

Installing the Java application

If the Java jars were not included in the same zip file as the user exit shared library, then extract the Java application into the Oracle GoldenGate installation directory, as shown above. There is no default location for the Java application; its installation location is defined by the by setting the java.class.path property in the user exit properties file:

```
javawriter.bootoptions=-Djava.class.path=ggjava/ggjava.jar
```

CHAPTER 3

VAM: Configuring Message Capture

Configuring the VAM Extract

To run the Java message capture application you need 1) the Oracle GoldenGate for Java adapter, 2) an Extract process, 3) an Extract parameter file configured for message capture and 4) a description of the incoming data format, such as a source definitions file.

Adding the Extract

To add the message capture VAM to the Oracle GoldenGate installation, add an Extract and the trail that it will create using GGSCI commands:

```
ADD EXTRACT jmsvam, VAM
ADD EXTTRAIL ./dirdat/id, EXTRACT jmsvam, MEGABYTES 100
```

The process name (`jmsvam`) can be replaced with any process name that is no more than 8 characters. The trail identifier (`id`) can be any two characters.

NOTE Commands to position the Extract, such as BEGIN or EXTRBA, are not supported for message capture. The Extract will always resume by reading messages from the end of the message queue.

Configuring the Extract parameters

The Extract parameter file contains the parameters needed to define and invoke the VAM. Sample Extract parameters for communicating with the VAM are shown in the table.

Parameter	Description
`EXTRACT jmsvam`	The name of the Extract process.
`VAM ggjava_vam.dll, PARAMS dirprm/jmsvam.properties`	Specifies the name of the VAM library and the location of the properties file. The VAM properties should be in the dirprm directory of the Oracle GoldenGate installation location.

Parameter	Description
`TRANLOGOPTIONS VAMCOMPATIBILITY 1`	Specifies use of the original implementation of the VAM.
`TRANLOGOPTIONS GETMETADATAFROMVAM`	Specifies that metadata will be sent by the VAM.
`EXTTRAIL dirdat/id`	Specifies the identifier of the target trail Extract creates.
`TABLE OGG.*`	A list of tables to process. Wildcards may be used in the filename.

Configuring the message capture

Message capture is configured by the properties in the VAM properties file. This file is identified by the PARAMS option of the Extract VAM parameter and used to determine logging characteristics, parser mappings and JMS connection settings.

Connecting and retrieving the messages

Connecting to JMS

Connectivity to JMS is through a generic JMS interface. Properties can be set to configure the following characteristics of the connection:

- Java classpath for the JMS client
- Name of the JMS queue or topic source destination
- Java Naming and Directory Interface (JNDI) connection properties
    - Connection properties for Initial Context
    - Connection factory name
    - Destination name
- Security information
    - JNDI authentication credentials
    - JMS user name and password

The Extract process that is configured to work with the VAM (such as the `jmsvam` in the example) will connect to the message system. when it starts up.

NOTE The Extract may be included in the Manger's AUTORESTART list so it will automatically be restarted if there are connection problems during processing.

Currently the Oracle GoldenGate for Java message capture adapter supports only JMS text messages.

Retrieving messages

The connection processing performs the following steps when asked for the next message:

- Start a local JMS transaction if one is not already started.
- Read a message from the message queue.
- If the read fails because no message exists, return an end-of-file message.
- Otherwise return the contents of the message.

Completing the transaction

Once all of the messages that make up a transaction have been successfully retrieved, parsed, and written to the Oracle GoldenGate trail, the local JMS transaction is committed and the messages removed from the queue or topic. If there is an error the local transaction is rolled back leaving the messages in the JMS queue.

CHAPTER 4

VAM: Parsing the message

Parsing overview

The role of the parser is to translate JMS text message data and header properties into an appropriate set of transactions and operations to pass into the VAM interface. To do this, the parser always needs to find certain data:

- Transaction identifier
- Sequence identifier
- Timestamp
- Table name
- Operation type
- Column data specific to a particular table name and operation type

Other data will be used if the configuration requires it:

- Transaction indicator
- Transaction name
- Transaction owner

The parser can obtain this data from JMS header properties, system generated values, static values, or in some parser specific way. This depends on the nature of the piece of information.

Parser types

The Oracle GoldenGate message capture adapter supports three types of parsers:

- Fixed – Messages contain data presented as fixed width fields in contiguous text.
- Delimited – Messages contain data delimited by field and end of record characters.
- XML – Messages contain XML data accessed through XPath expressions.

Source and target data definitions

There are several ways source data definitions can be defined using a combination of properties and external files. The Oracle GoldenGate Gendef utility generates a a standard source definitions file based on these data definitions and parser properties. The options vary based on parser type:

- Fixed – COBOL copybook, source definitions or user defined

- Delimited – source definitions or user defined
- XML – source definitions or user defined

There are several properties that configure how the selected parser gets data and how the source definitions are converted to target definitions.

Required Data

Transaction identifier

The transaction identifier (`txid`) groups operations into transactions when written to the Oracle GoldenGate trail file. The Oracle GoldenGate message capture adapter supports only contiguous, non-interleaved transactions. The transaction identifier can be any unique value that increases for each transaction. A system generated value can generally be used.

Sequence identifier

The sequence identifier (`seqid`) identifies each operation internally. This can be used during recovery processing to identify operations that have already been written to the Oracle GoldenGate trail. The sequence identifier can be any unique value that increases for each operation. The length should be fixed.

The JMS Message ID can be used as a sequence identifier if the message identifier for that provider increases and is unique. However, there are cases (e.g. using clustering, failed transactions) where JMS does not guarantee message order or when the ID may be unique but not be increasing. The system generated Sequence ID can be used, but it can cause duplicate messages under some recovery situations. The recommended approach is to have the JMS client that adds messages to the queue set the Message ID, a header property, or some data element to an application-generated unique value that is increasing.

Timestamp

The timestamp (`timestamp`) is used as the commit timestamp of operations within the Oracle GoldenGate trail. It should be increasing but this is not required, and it does not have to be unique between transactions or operations. It can be any date format that can be parsed.

Table name

The table name is used to identify the logical table to which the column data belongs. The adapter requires a two part table name in the form SCHEMA_NAME.TABLE_NAME. This can either be defined separately (`schema and table`) or as a combination of schema and table (`schemaandtable`).

A single field may contain both schema and table name, they may be in separate fields, or the schema may be included in the software code so only the table name is required. How the schema and table names can be specified depends on the parser. In any case the two part logical table name is used to write records in the Oracle GoldenGate trail and to generate the source definitions file that describes the trail.

Operation type

The operation type (`optype`) is used to determine whether an operation is an insert, update

or delete when written to the Oracle GoldenGate trail. The operation type value for any specific operation is matched against the values defined for each operation type.

The data written to the Oracle GoldenGate trail for each operation type depends on the Extract configuration:

- Inserts
  - The after values of all columns are written to the trail.
- Updates
  - Default – The after values of keys are written. The after values of columns that have changed are written if the before values are present and can be compared. If before values are not present then all columns are written.
  - NOCOMPRESSUPDATES – The after values of all columns are written to the trail.
  - GETUPDATEBEFORES – The before and after values of columns that have changed are written to the trail if the before values are present and can be compared. If before values are not present only after values are written.
  - If both NOCOMPRESSUPDATES and GETUPDATEBEFORES are included, the before and after values of all columns are written to the trail if before values are present
- Deletes
  - Default – The before values of all keys are written to the trail.
  - NOCOMPRESSDELETES – The before values of all columns are written to the trail.

Primary key update operations may also be generated if the before values of keys are present and do not match the after values.

Column data

All parsers retrieve column data from the message text and write it to the Oracle GoldenGate trail. In some cases the columns are read in index order as defined by the source definitions, in other cases they are accessed by name.

Depending on the configuration and original message text, both before and after or only after images of the column data may be available. For updates, the data for non-updated columns may or may not be available.

All column data is retrieved as text. It is converted internally into the correct data type for that column based on the source definitions. Any conversion problem will result in an error and the process will abend.

Optional data

Transaction indicator

The relationship of transactions to messages can be:

- One transaction per message

  This is determined automatically by the scope of the message.
- Multiple transactions per message

This is determined by the transaction indicator (`txind`). If there is no transaction indicator, the XML parser can create transactions based on a matching transaction rule.

- Multiple messages per transaction

  The transaction indicator (`txind`) is required to specify whether the operation is the beginning, middle, end or the whole transaction. The transaction indicator value for any specific operation is matched against the values defined for each transaction indicator type. A transaction is started if the indicator value is beginning or whole, continued if it is middle, and ended if it is end or whole.

Transaction name

The transaction name (`txname`) is optional data that can be used to associate an arbitrary name to a transaction. This can be added to the trail as a token using a GETENV function.

Transaction owner

The transaction owner (txowner) is optional data that can be used to associate an arbitrary user name to a transaction. This can be added to the trail as a token using a GETENV function, or used to exclude certain transactions from processing using the EXCLUDEUSER Extract parameter.

Fixed width parsing

Fixed width parsing is based on a data definition that defines the position and the length of each field. This is in the format of a Cobol copybook. A set of properties define rules for mapping the copybook to logical records in the Oracle GoldenGate trail and in the source definitions file.

The incoming data should consist of a standard format header followed by a data segment. Both should contain fixed width fields. The data is parsed based on the PIC definition in the copybook. It is written to the trail translated as explained in "Header and record data type translation" on page 27.

Header

The header must be defined by a copybook 01 level record that includes the following:

- A commit timestamp or a change time for the record
- A code to indicate the type of operation: insert, update, or delete
- The copybook record name to use when parsing the data segment

Any fields in the header record that are not mapped to Oracle GoldenGate header fields are output as columns.

Specifying a header

The following example shows a copybook definition containing the required header values:

```
01 HEADER.
   20 Hdr-Timestamp          PIC X(23)
   20 Hdr-Source-DB-Function PIC X
   20 Hdr-Source-DB-Rec-ID    PIC X(8)
```

You would set the following properties for this example:

```
fixed.header=HEADER
fixed.timestamp=Hdr-Timestamp
fixed.optype=Hdr-Source-DB-Function
fixed.table=Hdr-Source-DB-Rec-Id
```

The logical name table output in this case will be the value of `Hdr-Source-DB-Rec-Id`.

Specifying compound table names

More than one field can be used for a table name. For example, you can define the logical schema name through a static property such as:

```
fixed.schema=MYSCHEMA
```

Then you can add a property that defines the data record as multiple fields from the copybook header definition.

```
01  HEADER.
    20  Hdr-Source-DB              PIC X(8).
    20  Hdr-Source-DB-Rec-Id       PIC X(8).
    20  Hdr-Source-DB-Rec-Version  PIC 9(4).
    20  Hdr-Source-DB-Function     PIC X.
    20  Hdr-Timestamp              PIC X(22).
```

```
fixed.header=HEADER
fixed.table=Hdr-Source-DB-Rec-Id,Hdr-Source-DB-Rec-Version
fixed.schema=MYSCHEMA
```

The fields will be concantenated to result in logical schema and table names of the form:

```
MYSCHEMA.Hdr-Source-DB-Rec-Id+Hdr-Source-DB-Rec-Version
```

Specifying timestamp formats

A timestamp is parsed using the default format YYYY-MM-DD HH:MM:SS.FFF, with FFF depending on the size of the field.

Specify different incoming formats by entering a comment before the datetime field as shown in the next example.

```
01  HEADER.
* DATEFORMAT YYYY-MM-DD-HH.MM.SS.FF
    20  Hdr-Timestamp      PIC X(23)
```

Specifying the function

Use properties to map the standard Oracle GoldenGate operation types to the optype values. The following example specifies that the operation type is in the `Hdr-Source-DB-Function` field and that the value for insert is A, update is U and delete is D.

```
fixed.optype=Hdr-Source-DB-Function
fixed.optype.insert=A
fixed.optype.update=U
fixed.optype.delete=D
```

Header and record data type translation

The data in the header and the record data are written to the trail based on the translated data type.

- A field definition preceded by a date format comment is translated to an Oracle GoldenGate datetime field of the specified size. If there is no date format comment, the field will be defined by its underlying datatype.
- A PIC X field is translated to the CHAR datatype of the indicated size.
- A PIC 9 field is translated to a NUMBER datatype with the defined precision and scale. Numbers that are signed or unsigned and those with or without decimals are supported.

The following examples show the translation for various PIC definitions.

Input	Output
`PIC XX`	`CHAR(2)`
`PIC X(16)`	`CHAR(16)`
`PIC 9(4)`	`NUMBER(4)`
`* YYMMDD` `PIC 9(6)`	`DATE(10)` `YYYY-MM-DD`
`PIC 99.99`	`NUMBER(4,2)`
`PIC 9(5)V99`	`NUMBER(7,2)`

In the example an input YYMMDD date of 100522 is translated to 2010-05-22. The number 1234567 with the specified format PIC 9(5)V99 is translated to a seven digit number with two decimal places, or 12345.67.

Key identification

A comment is used to identify key columns within the data record. The Gendef utility that generates the source definitions uses the comment to locate a key column.

In the following example `Account` has been marked as a key column for `TABLE1`.

```
01 TABLE1
* KEY
20  Account     PIC X(19)
20  PAN_Seq_Num PIC 9(3)
```

Delimited parsing

Delimited parsing is based a preexisting source definitions files and a set of properties. The properties specify the delimiters to use and other rules, such as whether there are column names and before values. The source definitions file determines the valid tables to be processed and the order and datatype of the columns in the tables.

The format of the delimited message is:

$\{METACOLS\}^n[,\{COLNAMES\}]^m[,\{COLBEFOREVALS\}]^m,\{COLVALUES\}^m$\n

Where: There can be *n* metadata columns each followed by a field delimiter such as the comma shown in the format statement.

There can be *m* column values. Each of these are preceded by a field delimiter such as a comma.

The column name and before value are optional.

Each record is terminated by an end of line delimiter, such as `\n`.

Metadata columns

The metadata columns correspond to the header and contain fields that have special meaning. Metadata columns should include the following information.

- **optype** contains values indicating if the record is an insert, update, or delete. The default values are I, U, and D.
- **timestamp** indicates type of value to use for the commit timestamp of the record. The format of the timestamp defaults to YYYY-DD-MM HH:MM:SS.FFF.
- **schemaandtable** is the full table name for the record in the format SCHEMA.TABLE.
- **schema** is the record's schema name.
- **table** is the record's table name.
- **txind** is a value that indicates whether the record is the beginning, middle, end or the only record in the transaction. The default values are 0, 1, 2, 3.
- **id** is the value used as the sequence number (RSN or CSN) of the record. The id of the first record (operation) in the transaction is used for the sequence number of the transaction.

Parsing properties

Properties to describe delimiters

The following properties determine the parsing rules for delimiting the record.

- **fielddelim** specifies one or more ASCII or hexadecimal characters as the value for the field delimiter
- **recorddelim** specifies one or more ASCII or hexadecimal characters as the value for the record delimiter
- **quote** specifies one or more ASCII or hexadecimal characters to use for quoted values
- **nullindicator** specifies one or more ASCII or hexadecimal characters to use for NULL values

You can define escape characters for the delimiters so they will be replaced if the characters are found in the text. For example if a backslash and apostrophe (\') are specified, then the input "They used Mike\'s truck" is translated to "They used Mike's truck". Or if two quotes ("") are specified, "They call him ""Big Al""" is translated to "They call him "Big Al"".

Data values may be present in the record without quotes, but the system only removes escape characters within quoted values. A non-quoted string that matches a null indicator is treated as null.

Properties to describe values

The following properties provide more information:

- **hasbefores** indicates before values are present for each record
- **hasnames** indicates column names are present for each record
- **afterfirst** indicates column after values come before column before values
- **isgrouped** indicates all column names, before values and after values are grouped together in three blocks, rather than alternately per column

Properties to describe date and time

The default format YYYY-DD-MM HH:MM:SS.FFF is used to parse dates. The user can use properties to override this on a global, table or column level. Examples of changing the format are shown below.

```
delim.dateformat.default=MM/DD/YYYY-HH:MM:SS
delim.dateformat.MY.TABLE=DD/MMM/YYYY
delim.dateformat.MY.TABLE.COL1=MMYYYY
```

Parsing steps

The steps in delimited parsing are:

1. The parser first reads and validates the metadata columns for each record.
2. This provides the table name, which can then be used to look up column definitions for that table in the source definitions file.
3. If a definition cannot be found for a table, the processing will stop.
4. Otherwise the columns are parsed and output to the trail in the order and format defined by the source definitions.

XML parsing

XML parsing is based on a preexisting source definitions file and a set of properties. The properties specify rules to determine XML elements and attributes that correspond to transactions, operations and columns. The source definitions file determines the valid tables to be processed and the ordering and data types of columns in those tables.

Styles of XML

The XML message is formatted in either dynamic or static XML. At runtime the contents of *dynamic XML* are data values that cannot be predetermined using a sample XML or XSD document. The contents of *static XML* that determine tables and column element or attribute names can be predetermined using those sample documents.

The following two examples contain the same data.

An example of static XML

```
<NewMyTableEntries>
  <NewMyTableEntry>
    <CreateTime>2010-02-05:10:11:21</CreateTime>
    <KeyCol>keyval</KeyCol>
    <Col1>col1val</Col1>
  </NewMyTableEntry>
</NewMyTableEntries>
```

The `NewMyTableEntries` element marks the transaction boundaries. The `NewMyTableEntry` indicates an insert to MY.TABLE. The timestamp is present in an element text value, and the column names are indicated by element names.

You can define rules in the properties file to parse either of these two styles of XML through a set of XPath-like properties. The goal of the properties is to map the XML to a predefined source definitions file through XPath matches.

An example of dynamic XML

```
<transaction id="1234" ts="2010-02-05:10:11:21">
  <operation table="MY.TABLE" optype="I">
    <column name="keycol" index="0">
      <aftervalue><![CDATA[keyval]]></aftervalue>
    </column>
    <column name="col1" index="1">
      <aftervalue><![CDATA[col1val]]></aftervalue>
    </column>
  </operation>
</transaction>
```

Every operation to every table has the same basic message structure consisting of transaction, operation and column elements. The table name, operation type, timestamp, column names, column values, etc. are obtained from attribute or element text values.

XML parsing rules

Independent of the style of XML, the parsing process needs to determine:

- Transaction boundaries
- Operation entries and metadata including:
  - Table name
  - Operation type
  - Timestamp
- Column entries and metadata including:
  - Either the column name or index; if both are specified the system will check to see if the column with the specified data has the specified name.
  - Column before or after values, sometimes both.

This is done through a set of interrelated rules. For each type of XML message that is to be processed you name a rule that will be used to obtain the required data. For each of these named rules you add properties to:

- Specify the rule as a transaction, operation, or column rule type. Rules of any type are required to have a specified name and type.
- Specify the XPath expression to match to see if the rule is active for the document being processed. This is optional; if not defined the parser will match the node of the parent rule or the whole document if this is the first rule.
- List detailed rules (subrules) that are to be processed in the order listed. Which subrules are valid is determined by the rule type. Subrules are optional.

In the following example the top-level rule is defined as `genericrule`. It is a `transaction` type rule. Its subrules are defined in `oprule` and they are of the type `operation`.

```
xmlparser.rules=genericrule
xmlparser.rules.genericrule.type=tx
xmlparser.rules.genericrule.subrules=oprule
xmlparser.rules.oprule.type=op
```

XPath expressions

The XML parser supports a subset of XPath expressions necessary to match elements and extract data. An expression can be used to match a particular element or to extract data.

When doing data extraction most of the path is used to match. The tail of the expression is used for extraction.

Supported constructs:

/e	Use the absolute path from the root of the document to match e.
./e or e	Use the relative path from current node being processed to match e.
../e	Use a path based on the parent of the current node (can be repeated) to match e.
//e	Match e wherever it occurs in a document.
*	Match any element. Note: Partially wild-carded names are not supported.
[n]	Match the nth occurrence of an expression.
[x=v]	match when x is equal to some value v where x can be: ◆ @att – some attribute value ◆ text() – some text value ◆ name() the element name ◆ position() the element position

Supported expressions

Match root element	/My/Element
Match sub element to current node	./Sub/Element
Match nth element	/My/*[n]
Match nth Some element	/My/Some[n]

Match any text value	/My/*[text() ='value']
Match the text in Some element	/My/Some[text() = 'value']
Match any attribute	/My/*[@att = 'value']
Match the attribute in Some element	/My/Some[@att = 'value']

Obtaining data values

In addition to matching paths, the XPath expressions can also be used to obtain data values, either absolutely or relative to the current node being processed. Data value expressions can contain any of the path elements above, but must end with one of the value accessors listed below.

@att	Some attribute value.
text()	The text content (value) of an element.
content()	The full content of an element, including any child XML nodes.
name()	The name of an element.
position()	The position of an element in its parent.

Some examples:

To extract the relative element text value:

```
/My/Element/text()
```

To extract the absolute attribute value:

```
/My/Element/@att
```

To extract element text value with a match:

```
/My/Some[@att = 'value']/Sub/text()
```

NOTE Path accessors, such as ancestor/descendent/self, are not supported.

Other value expressions

The values extracted by the XML parser are either column values or properties of the transaction or operation, such as table or timestamp. These values are either obtained from XML using XPath or through properties of the JMS message, system values, or hard coded values. The XML parser properties specify which of these options are valid for obtaining the values for that property.

The following example specifies that `timestamp` can be an XPath expression, a JMS property, or the system generated timestamp.

```
{txrule}.timestamp={xpath-expression}|${jms-property}|*ts
```

The next example specifies that `table` can be an XPath expression, a JMS property, or hard coded value.

```
{oprule}.table={xpath-expression}|${jms-property}|"value"
```

The last example specifies that `name` can be a XPath expression or hard coded value.

```
{colrule}.timestamp={xpath-expression}|"value"
```

Transaction rules

The rule that specifies the boundary for a transaction is at the highest level. Messages may 1) contain a single transaction, 2) contain multiple transactions, or 3) a transaction may span messages. These are specified as follows:

- **single** - The transaction rule match is not defined.
- **multiple** - Each transaction rule match defines new transaction.
- **span** – No transaction rule is defined; instead a transaction indicator is specified in an operation rule.

For a transaction rule, the following properties of the rule may also be defined through XPath or other expressions:

- **timestamp** – The time at which the transaction occurred.
- **txid** – The identifier for the transaction.

Transaction rules can have multiple subrules, but each must be of type operation.

Examples

The following example specifies a transaction that is the whole message and includes a timestamp that comes from the JMS property.

```
singletxrule.timestamp=$JMSTimeStamp
```

The following example matches the root element transaction and obtains the timestamp from the `ts` attribute.

```
dyntxrule.match=/Transaction
dyntxrule.timestamp=@ts
```

Operation rules

An operation rule can either be a subrule of a transaction rule, or a highest level rule (if the transaction is a property of the operation).

In addition to the standard rule properties, an operation rule should also define the following through XPath or other expressions:

- **timestamp** – The timestamp of the operation. This is optional the transaction rule is defined.
- **table** – The name of the table on which this is an operation. Use this with schema.
- **schema** – The name of schema for the table.
- **schemaandtable** – Both schema and table name together in the form SCHEMA.TABLE. This can be used in place of the individual table and schema properties.
- **optype** – Specifies whether this is an insert, update or delete operation based on optype values:
    - **optype.insertval** – The value indicating an insert. The default is I.

- **optype.updateval** – The value indicating an update. The default is U.
- **optype.deleteval** – The value indicating a delete. The default is D.

- **seqid** – The identifier for the operation. This will be the transaction identifier if `txid` has not already been defined at the transaction level.
- **txind** – Specifies whether this operation is the beginning of a transaction, in the middle or at the end; or if it is the whole operation. This jproperty is optional and not valid if the operation rule is a subrule of a transaction rule.

Operation rules can have multiple subrules of type operation or column.

Examples

The following example dynamically obtains operation information from the /Operation element of a /Transaction.

```
dynoprule.match=./Operation
dynoprule.schemaandtable=@table
dynoprule.optype=@type
```

The following example statically matches /NewMyTableEntry element to an insert operation on the MY.TABLE table.

```
statoprule.match=./NewMyTableEntry
statoprule.schemaandtable="MY.TABLE"
statoprule.optype="I"
statoprule.timestamp=./CreateTime/text()
```

Column rules

A column rule must be a subrule of an operation rule. In addition to the standard rule properties, a column rule should also define the following through XPath or other expressions.

- **name** – The name of the column within the table definition.
- **index** – The index of the column within the table definition.

NOTE If only one of the `name` and `index` is defined, the other will be determined.

- **before.value** – The before value of the column. This is required for deletes, but is optional for updates.
- **before.isnull** – Indicates whether the before value of the column is null.
- **before.ismissing** – Indicates whether the before value of the column is missing.
- **after.value** – The before value of the column. This is required for deletes, but is optional for updates.
- **after.isnull** – Indicates whether the before value of the column is null.
- **after.ismissing** – Indicates whether the before value of the column is missing.
- **value** – An expression to use for both `before.value` and `after.value` unless overridden by specific before or after values. Note that this does not support different before values for updates.
- **isnull** – An expression to use for both before.isnull and after.isnull unless overridden.
- **ismissing** – An expression to use for both before.ismissing and after.ismissing unless overridden.

Examples

The following example dynamically obtains column information from the `/Column` element of an `/Operation`.

```
dyncolrule.match=./Column
dyncolrule.name=@name
dyncolrule.before.value=./beforevalue/text()
dyncolrule.after.value=./aftervalue/text()
```

The following example statically matches the `/KeyCol` and `/Col1` elements to columns in MY.TABLE.

```
statkeycolrule.match=/KeyCol
statkeycolrule.name=”keycol”
statkeycolrule.value=./text()
statcol1rule.match=/Col1
statcol1rule.name=”col1”
statcol1rule.value=./text()
```

Overall rules example

The following example uses the XML samples shown earlier with appropriate rules to enerate the same resulting operation on the MY.TABLE table.

Dynamic XML

```
<transaction id="1234"
    ts="2010-02-05:10:11:21">
  <operation table="MY.TABLE" optype="I">
    <column name="keycol" index="0">
      <aftervalue>
<![CDATA[keyval]]>
      </aftervalue>
    </column>
    <column name="col1" index="1">
      <aftervalue>
         <![CDATA[col1val]]>
      </aftervalue>
    </column>
  </operation>
</transaction>
```

```
dyntxrule.match=/Transaction
dyntxrule.timestamp=@ts
dyntxrule.subrules=dynoprule
dynoprule.match=./Operation
dynoprule.schemaandtable=@table
dynoprule.optype=@type
dynoprule.subrules=dyncolrule
dyncolrule.match=./Column
dyncolrule.name=@name
dyncolrule.before.value=./beforevalue/text()
dyncolrule.after.value=./aftervalue/text()
```

Static XML

```
NewMyTableEntries>
  <NewMyTableEntry>
    <CreateTime>
       2010-02-05:10:11:21
    </CreateTime>
    <KeyCol>keyval</KeyCol>
    <Col1>col1val</Col1>
  </NewMyTableEntry>
</NewMyTableEntries>
```

```
stattxrule.match=/NewMyTableEntries
stattxrule.subrules= statoprule
statoprule.match=./NewMyTableEntry
statoprule.schemaandtable=”MY.TABLE”
statoprule.optype=”I”
statoprule.timestamp=./CreateTime/text()
statoprule.subrules= statkeycolrule,
statcol1rule
statkeycolrule.match=/KeyCol
statkeycolrule.name=”keycol”
statkeycolrule.value=./text()
statcol1rule.match=/Col1
statcol1rule.name=”col1”
statcol1rule.value=./text()
```

```
INSERT INTO MY.TABLE (KEYCOL, COL1)
VALUES ('keyval', 'col1val')
```

Source definitions generation utility

Oracle GoldenGate for Java includes a Gendef utility that generates an Oracle GoldenGate source definitions file from the properties defined in a properties file. It creates a normalized definition of tables based on the property settings and other parser-specific data definition values.

The syntax to run this utility is:

```
gendef –prop {property file} [-out {output file}]
```

This defaults to sending the source definitions to standard out, but it can be directed to a file using the –out parameter. For example:

```
gendef –prop dirprm/jmsvam.properties -out dirdef/msgdefs.def
```

The output source definitions file can then be used in a pump or delivery process to interpret the trail data created through the VAM.

CHAPTER 5
VAM: Message Capture Properties

This section explains the options available for configuration of the property file for the Oracle GoldenGate for Java VAM.

Place this property file in the /dirprm directory of your Oracle GoldenGate installation location. The name of the VAM properties file is set with the Extract VAM parameter.

All properties in the property file are of the form: fully.qualified.name=value. The value may be integer, boolean, single string, or comma delimited strings.

Comments can be entered in the properties file with the # prefix at the beginning of the line. For example:

```
# This is a property comment
some.property=value
```

Properties themselves can also be commented out. However you cannot place a comment at the end of a line; either the whole line is a comment or it is a property.

Logging and connection properties

Logging properties

Logging is controlled by the following properties.

log.logname

Specifies the prefix to the log file name. This must be a valid ASCII string. The log file name has the current date appended to it, in yyyymmdd format, together with the .log extension.

The following example will create a log file of name writer_20100803.log on August 3, 2010. The log file will roll over each day, independent of the stopping and starting of the process.

```
# log file prefix
log.logname=writer
```

The following example will create a log file of name msgv_20100803.log on August 3, 2010.

```
# log file prefix
log.logname=msgv
```

log.level

Specifies the overall log level for all modules. The syntax is:

```
log.level=ERROR|WARN|INFO|DEBUG
```

The log levels are defined as follows:

ERROR – Only write messages if errors occur

WARN – Write error and warning messages

INFO – Write error, warning and informational messages

DEBUG – Write all messages, including debug ones.

The default logging level is INFO. The messages in this case will be produced on startup, shutdown and periodically during operation. If the level is switched to DEBUG, large volumes of messages may occur which could impact performance. For example, the following sets the global logging level to INFO:

```
# global logging level
log.level=INFO
```

log.tostdout

Controls whether or not log information is written to standard out. This setting is useful if the Extract process is running with a VAM started from the command line or on an operating system where `stdout` is piped into the report file. Generally Oracle GoldenGate processes run as background processes however.

The syntax is:

```
goldengate.log.tostdout=true|false
```

The default is false.

log.tofile

Controls whether or not log information is written to the specified log file.The syntax is:

```
log.tofile=true|false
```

The default is false. Log output is written to the specified log file when set to true.

log.modules, log.level.{module}

Specifies the log level of the individual source modules that comprise the user exit. This is typically used only for advanced debugging. It is possible to increase the logging level to DEBUG on a per module basis to help troubleshoot issues. The default levels should not be changed unless asked to do so by Oracle GoldenGate support.

JMS connection properties

The JMS connection properties set up the connection, such as how to start up the JVM for JMS integration.

jvm.boot options

Specifies the classpath and boot options that will be applied when the user exit starts up the JVM. The path needs colon (:) separators for UNIX/Linux and semicolons (;) for Windows.

The syntax is:

```
jvm.bootoptions={option}[, ...]
```

The options are the same as those passed to Java executed from the command line. They may include classpath, system properties, and JVM memory options (such as max memory or initial memory) that are valid for the version of Java being used. Valid options may vary based on the JVM version and provider.

For example (all on a single line):

```
jvm.bootoptions= -Djava.class.path=ggjava/ggjava.jar
             -Dlog4j.configuration=my-log4j.properties
```

The log4j.configuration property could be a fully qualified URL to a log4j properties file; by default this file is searched for in the classpath. You may use your own log4j configuration, or one of the pre-configured log4j settings: log4j.properties (default level of logging), debug_log4j.properties (debug logging) or trace_log4j.properties (very verbose logging).

jms.report.output

Specifies where the JMS report is written. The syntax is:

```
jms.report.output=report|log|both
```

Where: `report` sends the JMS report to the Oracle GoldenGate report file. This is the default.

`log` will write to the Java log file (if one is configured)

`both` will send to both locations.

jms.report.time

Specifies the frequency of report generation based on time.

```
jms.report.time={time-specification}
```

The following examples write a report every 30 seconds, 45 minutes and eight hours.

```
jms.report.time=30sec
jms.report.time=45min
jms.report.time=8hr
```

jms.report.records

Specifies the frequency of report generation based on number of records.

```
jms.report.records={number}
```

The following example writes a report every 1000 records.

```
jms.report.records=1000
```

jms.id

Specifies that a unique identifier with the indicated format is passed back from the JMS integration to the message capture VAM. This may be used by the VAM as a unique sequence ID for records.

```
jms.id=ogg|time|wmq|activemq|{message-header}|{custom-java-class}
```

Where: `ogg` returns the message header property GG_ID which is set by Oracle GoldenGate JMS delivery.

`time` uses a system timestamp as a starting point for the message ID

`wmq` reformats a WebSphere MQ Message ID for use with the VAM

`activemq` reformats an ActiveMQ Message ID for use with the VAM

`{message-header}` specifies the user customized JMS message header to be included, such as JMSMessageID, JMSCorrelationID, or JMSTimestamp.

`{custom-java-class}` specifies a custom Java class that creates a string to be used as an ID.

For example:

```
jms.id=time
jms.id=JMSMessageID
```

The ID returned must be unique, incrementing, and fixed-width. If there are duplicate numbers, the duplicates are skipped. If the message ID changes length, the Extract process will abend.

jms.destination

Specifies the queue or topic name to be looked up via JNDI.

```
jms.destination={jndi-name}
```

For example:

```
jms.destination=sampleQ
```

jms.connectionFactory

Specifies the connection factory name to be looked up via JNDI.

```
jms.connectionFactory={jndi-name}
```

For example

```
jms.connectionFactory=ConnectionFactory
```

jms.user, jms.password

Sets the user name and password of the JMS connection, as specified by the JMS provider.

```
jms.user={user-name}
jms.password={password}
```

This is not used for JNDI security. To set JNDI authentication, see the JNDI `java.naming.security` properties.

For example:

```
jms.user=myuser
jms.password=mypasswd
```

JNDI properties

In addition to specific properties for the message capture VAM, the JMS integration also supports setting JNDI properties required for connection to an Initial Context to look up the connection factory and destination. The following properties must be set:

```
java.naming.provider.url={url}
java.naming.factory.initial={java-class-name}
```

If JNDI security is enabled, the following properties may be set:

```
java.naming.security.principal={user-name}
java.naming.security.credentials={password-or-other-authenticator}
```

For example:

```
java.naming.provider.url= t3://localhost:7001
java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
java.naming.security.principal=jndiuser
java.naming.security.credentials=jndipw
```

Parser properties

Properties specify the formats of the message and the translation rules for each type of parser: fixed, delimited, or XML. Set the `parser.type` property to specify which parser to use. The remaining properties are parser specific.

Setting the type of parser

parser.type

Specifies the parser to use.

```
parser.type=fixed|delim|xml
```

Where: `fixed` invokes the fixed width parser

`delim` invokes the delimited parser

`xml` invokes the XML parser

For example:

```
parser.type=delim
```

Fixed parser properties

The following properties are required for the fixed parser.

fixed.schema

Specifies the type of file used as metadata for message capture. The two valid options are sourcedefs and copybook.

```
fixed.schematyype=sourcedefs|copybook
```

For example:

```
fixed.schematype=copybook
```

The value of this property determines the other properties that must be set in order to successfully parse the incoming data.

fixed.sourcedefs

If the `fixed.schematype=sourcedefs`, this property specifies the location of the source definitions file that is to be used.

```
fixed.sourcedefs={file-location}
```

For example:

```
fixed.sourcedefs=dirdef/hrdemo.def
```

fixed.copybook

If the `fixed.schematype=copybook`, this property specifies the location of the copybook file to be used by the message capture process.

```
fixed.copybook={file-location}
```

For example:

```
fixed.copybook=test_copy_book.cpy
```

fixed.header

Specifies the name of the sourcedefs entry or copybook record that contains header information used to determine the data block structure:

```
fixed.header={record-name}
```

For example:

```
fixed.header=HEADER
```

fixed.seqid

Specifies the name of the header field, JMS property, or system value that contains the `seqid` used to uniquely identify individual records. This value must be continually incrementing and the last character must be the least significant.

```
fixed.seqid={field-name}|${jms-property}|*seqid
```

Where: `field-name` indicates the name of a header field containing the seqid

`jms-property` uses the value of the specified JMS header property. A special value of this is `$jmsid` which uses the value returned by the mechanism chosen by the `jms.id` property

`seqid` indicates a simple incrementing 64-bit integer generated by the system

For example:

```
fixed.seqid=$jmsid
```

fixed.timestamp

Specifies the name of the field, JMS property, or system value that contains the timestamp.

```
fixed.timestamp={field-name}|${jms-property}|*ts
```

For example:

```
fixed.timestamp=TIMESTAMP
fixed.timestamp=$JMSTimeStamp
fixed.timestamp=*ts
```

fixed.timestamp.format

Specifies the format of the timestamp field.

```
fixed.timestamp.format={format}
```

Where the format can include punctuation characters plus:

YYYY – four digit year

YY – two digit year

M[M] – one or two digit month

D[D] – one or two digit day

HH – hours in twenty four hour notation

MI – minutes

SS – seconds

Fn – n number of fractions

The default format is "YYYY-MM-DD:HH:MI:SS.FFF"

For example:

```
fixed.timestamp.format=YYYY-MM-DD-HH.MI.SS
```

fixed.txid

Specifies the name of the field, JMS property, or system value that contains the `txid` used to uniquely identify transactions. This value must increment for each transaction.

```
fixed.txid={field-name}|${jms-property}|*txid
```

For most cases using the system value of `*txid` is preferred.

For example:

```
fixed.txid=$JMSTxId
fixed.txid=*txid
```

fixed.txowner

Specifies the name of the field, JMS property, or static value that contains a user name associated with a transaction. This value may be used to exclude certain transactions from processing. This is an optional property.

```
fixed.txowner={field-name}|${jms-property}|"{value}"
```

For example:

```
fixed.txowner=$MessageOwner
fixed.txowner="jsmith"
```

fixed.txname

Specifies the name of the field, JMS property, or static value that contains an arbitrary name to be associated with a transaction. This is an optional property.

```
fixed.txname={field-name}|${jms-property}|"value"
```

For example:

```
fixed.txname="fixedtx"
```

fixed.optype

Specifies the name of the field, or JMS property that contains the operation type, which is validated against the `fixed.optype` values specified in the next sections.

```
fixed.header.optype={field-name}|${jms-property}
```

For example:

```
fixed.header.optype=FUNCTION
```

fixed.optype.insertval

This value identifies an insert operation. The default is `I`.

```
fixed.optype.insertval={value}|\x{hex-value}
```

For example:

```
fixed.optype.insertval=A
```

fixed.optype.updateval

This value identifies an update operation. The default is `U`.

```
fixed.optype.updateval={value}|\x{hex-value}
```

For example:

```
fixed.optype.updateval=M
```

fixed.optype.deleteval

This value identifies a delete operation.The default is `D`.

```
fixed.optype.deleteval={value}|\x{hex-value}
```

For example:

```
fixed.optype.deleteval=R
```

fixed.table

Specifies the name of the table. This enables the parser to find the data record definition needed to translate the non-header data portion.

```
fixed.table={field-name}|${jms-property}[,…]
```

More than one comma delimited field name may be used to determine the name of the table Each field name corresponds to a field in the header record defined by the `fixed.header` property or JMS property. The values of these fields are concatenated to identify the data record.

For example:

```
fixed.table=$JMSTableName
fixed.table=SOURCE_Db,SOURCE_Db_Rec_Version
```

fixed.schema

Specifies the static name of the schema when generating SCHEMA.TABLE table names.

```
fixed.schema=”{value}”
```

For example:

```
fixed.schema=”OGG”
```

fixed.txind

Specifies the name of the field or JMS property that contains a transaction indicator that is validated against the transaction indicator values. If this is not defined all operations within a single message will be seen to have occurred within a whole transaction. If defined, then it determines the beginning, middle and end of transactions. Transactions defined in this way can span messages. This is an optional property.

```
fixed.txind={field-name}|${jms-property}
```

For example:

```
fixed.txind=$TX_IND
```

fixed.txind.beginval

This value identifies an operation as the beginning of a transaction. The defaults is B.

```
fixed.txind.beginval={value}|\x{hex-value}
```

For example:

```
fixed.txind.beginval=0
```

fixed.txind.middleval

This value identifies an operation as the middle of a transaction. The default is M.

```
fixed.txind.middleval={value}|\x{hex-value}
```

For example:

```
fixed.txind.middleval=1
```

fixed.txind.endval

This value identifies an operation as the end of a transaction. The default is E.

```
fixed.txind.endval={value}|\x{hex-value}
```

For example:

```
fixed.txind.endval=2
```

fixed.txind.wholeval

This value identifies an operation as a whole transaction. The default is W.

```
fixed.txind.wholeval={value}|\x{hex-value}
```

For example:

```
fixed.txind.wholeval=3
```

Delimited Parser Properties

The following properties are required for the delimited parser except where otherwise noted.

delim.sourcedefs

Specifies the location of the source definitions file to use.

```
delim.sourcedefs={file-location}
```

For example:

```
delim.sourcedefs=dirdef/hrdemo.def
```

delim.header

Specifies the list of values that come before the data and assigns names to each.

```
delim.header={name},[…]
```

The names must be unique. They can be referenced in other `delim` properties or wherever header fields can be used.

For example:

```
delim.header=optype, tablename, ts
delim.timestamp=ts
```

delim.seqid

Specifies the name of the header field, JMS property, or system value that contains the `seqid` used to uniquely identify individual records. This value must increment and the last character must be the least significant.

```
delim.seqid={field-name}|${jms-property}|*seqid
```

Where: `field-name` indicates the name of a header field containing the seqid

`jms-property` uses the value of the specified JMS header property, a special value of this is $jmsid which uses the value returned by the mechanism chosen by the jms.id property

`seqid` indicates a simple continually incrementing 64-bit integer generated by the system

For example:

```
delim.seqid=$jmsid
```

delim.timestamp

Specifies the name of the JMS property, header field, or system value that contains the timestamp.

```
delim.timestamp={field-name}|${jms-property}|*ts
```

For example:

```
delim.timestamp=TIMESTAMP
delim.timestamp=$JMSTimeStamp
delim.timestamp=*ts
```

delim.timestamp.format

Specifies the format of the timestamp field.

```
delim.timestamp.format={format}
```

Where the format can include punctuation characters plus:

YYYY – four digit year

YY – two digit year

M[M] – one or two digit month

D[D] – one or two digit day

HH – hours in twenty four hour notation

MI – minutes

SS – seconds

Fn – n number of fractions

The default format is "YYYY-MM-DD:HH:MI:SS.FFF"

For example:

```
delim.timestamp.format=YYYY-MM-DD-HH.MI.SS
```

delim.txid

Specifies the name of the JMS property, header field, or system value that contains the `txid` used to uniquely identify transactions. This value must increment for each transaction.

```
delim.txid={field-name}|${jms-property}|*txid
```

For most cases using the system value of `*txid` is preferred.

For example:

```
delim.txid=$JMSTxId
delim.txid=*txid
```

delim.txowner

Specifies the name of the JMS property, header field, or static value that contains an arbitrary user name associated with a transaction. This value may be used to exclude certain transactions from processing. This is an optional property.

```
delim.txowner={field-name}|${jms-property}|”{value}”
```

For example:

```
delim.txowner=$MessageOwner
delim.txowner=”jsmith”
```

delim.txname

Specifies the name of the JMS property, header field, or static value that contains an arbitrary name to be associated with a transaction. This is an optional property.

```
delim.txname={field-name}|${jms-property}|”value”
```

For example:

```
delim.txname=”fixedtx”
```

delim.optype

Specifies the name of the JMS property or header field that contains the operation type.

This is compared to the values for `delim.optype.insertval`, `delim.optype.updateval` and `delim.optype.deleteval` to determine the operation.

```
delim.optype={field-name}|${jms-property}
```

For example:

```
delim.optype=optype
```

delim.optype.insertval

This value identifies an insert operation. The default is I.

```
delim.optype.insertval={value}|\x{hex-value}
```

For example:

```
delim.optype.insertval=A
```

delim.optype.updateval

This value identifies an update operation. The default is U.

```
delim.optype.updateval={value}|\x{hex-value}
```

For example:

```
delim.optype.updateval=M
```

delim.optype.deleteval

This value identifies a delete operation. The default is D.

```
delim.optype.deleteval={value}|\x{hex-value}
```

For example:

```
delim.optype.deleteval=R
```

delim.schemaandtable

Specifies the name of the JMS property or header field that contains the schema and table name in the form SCHEMA.TABLE.

```
delim.schemaandtable={field-name}|${jms-property}
```

For example:

```
delim.schemaandtable=$FullTableName
```

delim.schema

Specifies the name of the JMS property, header field, or hard-coded value that contains the schema name.

```
delim.schema={field-name}|${jms-property}|"value"
```

For example:

```
delim.schema="OGG"
```

delim.table

Specifies the name of the JMS property or header field that contains the table name.

```
delim.table={field-name}|${jms-property}
```

For example:

```
delim.table=TABLE_NAME
```

delim.txind

Specifies the name of the JMS property or header field that contains the transaction indicator to be validated against `beginval`, `middleval`, `endval` or `wholeval`. All operations within a single message will be seen as within one transaction if this property is not set. If it is set it determines the beginning, middle and end of transactions. Transactions defined in this way can span messages . This is an optional property.

```
delim.txind={field-name}|${jms-property}
```

For example:

```
delim.txind=txind
```

delim.txind.beginval

The value that identifies an operation as the beginning of a transaction. The default is B.

```
delim.txind.beginval={value}|\x{hex-value}
```

For example:

```
delim.txind.beginval=0
```

delim.txind.middleval

The value that identifies an operation as the middle of a transaction. The default is M.

```
delim.txind.middleval={value}|\x{hex-value}
```

For example:

```
delim.txind.middleval=1
```

delim.txind.endval

The value that identifies an operation as the end of a transaction. The default is E.

```
delim.txind.endval={value}|\x{hex-value}
```

For example:

```
delim.txind.endval=2
```

delim.txind.wholeval

The value that identifies an operation as a whole transaction. The default is W.

```
delim.txind.wholeval={value}|\x{hex-value}
```

For example:

```
delim.txind.wholeval=3
```

delim.fielddelim

Specifies the delimiter value used to separate fields (columns) in the data. This value is defined through characters or hexadecimal values:

```
delim.fielddelim={value}|\x{hex-value}
```

For example:

```
delim.fielddelim=,
delim.fielddelim=\xc7
```

delim.linedelim

Specifies the delimiter value used to separate lines (records) in the data. This value is defined using characters or hexadecimal values.

```
delim.linedelim={value}|\x{hex-value}
```

For example:

```
delim.linedelim=||
delim.linedelim=\x0a
```

delim.quote

Specifies the value used to identify quoted data. This value is defined using characters or hexadecimal values.

```
delim.quote={value}|\x{hex-value}
```

For example:

```
delim.quote=”
```

delim.nullindicator

Specifies the value used to identify NULL data. This value is defined using characters or hexadecimal values.

```
delim.nullindicator={value}|\x{hex-value}
```

For example:

```
delim.nullindicator=NULL
```

delim.fielddelim.escaped

Specifies the value that indicates a true field delimiter is present in data. This field

delimiter is replaced with the `fielddelim.escaped` value.

```
delim.fielddelim.escaped={value}|\x{hex-value}
```

The following example specifies the comma as the field delimiter surrounded by the $ escape characters.

```
delim.fielddelim.escaped=$,$
```

delim.linedelim.escaped

Specifies the value that indicates a true line delimiter is present in data. This line delimiter is replaced with the `linedelim.escaped` value.

```
delim.linedelim.escaped={value}|\x{hex-value}
```

For example:

```
delim.linedelim.escaped=\x0affa0
```

delim.quote.escaped

Specifies the value that indicates a true quote is present in data. This quote value is replaced with the `quote.escaped` value.

delim.quote.escaped={value}|\x{hex-value}

For example:

delim.quote.escaped=""

delim.nullindicator.escaped

Specifies the value that indicates a true null indicator is present in data. This indicator is replaced with the `nullindicator.escaped` value.

```
delim.nullindicator.escaped={value}|\x{hex-value}
```

For example:

```
delim.nullindicator.escaped={NULL}
```

delim.hasbefores

Specifies whether before values are present in the data.

```
delim.hasbefores=true|false
```

The default is false. The parser expects to find before and after values of columns for all records if `delim.hasbefores` is set to true. The before values are used for updates and deletes, the after values for updates and inserts. The `afterfirst` property specifies whether the before images are before the after images or after them. If `delim.hasbefores` is false, then no before values are expected.

For example:

```
delim.hasbefores=true
```

delim.hasnames

Specifies whether column names are present in the data.

```
delim.hasnames=true|false
```

The default is false. If true, the parser expects to find column names for all records. The parser validates the column names against the expected column names. If false, no column names are expected.

For example:

```
delim.hasnames=true
```

delim.afterfirst

Specifies whether after values are positioned before or after the before values.

```
delim.afterfirst=true|false
```

The default is false. If true, the parser expects to find the after values before the before values. If false, the after values are before the before values.

For example:

```
delim.afterfirst=true
```

delim.isgrouped

Specifies whether the column names and before and after images should be expected grouped together for all columns or interleaved for each column.

```
delim.isgrouped=true|false
```

The default is false. If true, the parser expects find a group of column names (if `hasnames` is true), followed by a group of before values (if `hasbefores`), followed by a group of after values (the `afterfirst` setting will reverse the before and after value order). If false, the parser will expect to find a column name (if `hasnames`), before value (if `hasbefores`) and after value for each column.

For example:

```
delim.isgrouped=true
```

delim.dateformat

Specifies the date format for column data. The format used to parse the date is a subset of the formats used for `parser.timestamp.format`. This is specified at a global level, a table level or column level.

```
delim.dateformat={format}
delim.dateformat.{TABLE}={format}
delim.dateformat.{TABLE}.{COLUMN}={format}
```

Where: {`format`} is the format defined for `parser.timestamp.format`

{`TABLE`} is a fully qualified table name

{`COLUMN`} is a column of the specified table.

For example:

```
delim.dateformat=YYYY-MM-DD HH:MI:SS
delim.dateformat.MY.TABLE=DD/MM/YY-HH.MI.SS
delim.dateformat.MY.TABLE.EXP_DATE=YYMM
```

XML parser properties

xml.sourcedefs

Specifies the location of the source definitions file.

```
xml.sourcedefs={file-location}
```

For example:

```
xml.sourcedefs=dirdef/hrdemo.def
```

xml.rules

Specifies the list of XML rules for parsing a message and converting to transactions, operations and columns:

```
xml.rules={xml-rule-name}[,…]
```

The specified XML rules are processed in the order listed. All rules matching a particular XML document may result in the creation of transactions, operations and columns. The specified XML rules should be transaction or operation type rules.

For example:

```
xml.rules=dyntxrule, statoprule
```

{rulename}.type

Specifies the type of XML rule.

```
{rulename}.type=tx|op|col
```

Where: `tx` indicates a transaction rule

`op` indicates an operation rule

`col` indicates a column rule

For example:

```
dyntxrule.type=tx
statoprule.type=op
```

{rulename}.match

Specifies an XPath expression used to determine whether the rule is activated for a particular document or not.

```
{rulename}.match={xpath-expression}
```

If the XPath expression returns any nodes from the document, the rule matches and further processing occurs. If it does not return any nodes, the rule is ignored for that document.

The following example activates the `dyntxrule` if the document has a root element of `Transaction`

```
dyntxrule.match=/Transaction
```

Where `statoprule` is a subrule of `stattxtule`, the following example activates the `statoprule` if the parent rule's matching nodes have child elements of `NewMyTableEntry`.

```
statoprule.match=./NewMyTableEntry
```

{rulename}.subrules

Specifies a list of rule names to check for matches if the parent rule is activated by its match.

```
{rulename}.subrules={xml-rule-name}[,…]
```

The specified XML rules are processed in the order listed. All matching rules may result in the creation of transactions, operations and columns.

Valid subrules are determined by the parent type. Transaction rules can only have operation subrules. Operation rules can have operation or column subrules. Column rules cannot have subrules.

For example:

```
dyntxrule.subrules=dynoprule
statoprule.subrules=statkeycolrule, statcol1rule
```

{txrule}.timestamp

Controls the transaction timestamp by instructing the adapter to 1) use the transaction commit timestamp contained in a specified XPath expression or JMS property or 2) use the current system time. This is an optional property.

```
{txrule}.timestamp={xpath-expression}|${jms-property}|*ts
```

The timestamp for the transaction may be overridden at the operation level, or may only be present at the operation level. Any XPath expression must end with a value accessor such as `@att` or `text()`.

For example:

```
dyntxrule.timestamp=@ts
```

{txrule}.timestamp.format

Specifies the format of the timestamp field.

```
{txrule}.timestamp.format={format}
```

Where the format can include punctuation characters plus:

YYYY – four digit year

YY – two digit year

M[M] – one or two digit month

D[D] – one or two digit day

HH – hours in twenty four hour notation

MI – minutes

SS – seconds

Fn – n number of fractions

The default format is "YYYY-MM-DD:HH:MI:SS.FFF"

For example:

```
dyntxrule.timestamp.format=YYYY-MM-DD-HH.MI.SS
```

{txrule}.seqid

Specifies the `seqid` for a particular transaction. This can be used when there are multiple transactions per message. Determines the XPath expression, JMS property, or system value that contains the transactions `seqid`. Any XPath expression must end with a value accessor such as `@att` or `text()`.

```
{txrule}.seqid={xpath-expression}|${jms-property}|*seqid
```

For example:

```
dyntxrule.seqid=@seqid
```

{txrule}.txid

Specifies the XPath expression, JMS property, or system value that contains the `txid` used to unique identify transactions. This value must increment for each transaction.

```
{txrule}.txid={xpath-expression}|${jms-property}|*txid
```

For most cases using the system value of `*txid` is preferred.

For example:

```
dyntxrule.txid=$JMSTxId
dyntxrule.txid=*txid
```

{txrule}.txowner

Specifies the XPath expression, JMS property, or static value that contains an arbitrary user name associated with a transaction. This value may be used to exclude certain transactions from processing.

```
{txrule}.txowner={xpath-expression}|${jms-property}|"{value}"
```

For example:

```
dyntxrule.txowner=$MessageOwner
dyntxrule.txowner="jsmith"
```

{txrule}.txname

Specifies the XPath expression, JMS property, or static value that contains an arbitrary name to be associated with a transaction. This is an optional property.

```
{txrule}.txname={xpath-expression}|${jms-property}|"value"
```

For example:

```
dyntxrule.txname="fixedtx"
```

{oprule}.timestamp

Controls the operation timestamp by instructing the adapter to 1) use the transaction commit timestamp contained in a specified XPath expression or JMS property or 2) use the current system time. This is an optional property.

```
{oprule}.timestamp={xpath-expression}|${jms-property}|*ts
```

The timestamp for the operation will override a timestamp at the transaction level.

Any XPath expression must end with a value accessor such as `@att` or `text()`.

For example:

```
statoprule.timestamp=./CreateTime/text()
```

{oprule}.timestamp.format

Specifies the format of the timestamp field.

```
{oprule}.timestamp.format={format}
```

Where the format can include punctuation characters plus:

YYYY – four digit year

YY – two digit year

M[M] – one or two digit month

D[D] – one or two digit day

HH – hours in twenty four hour notation

MI – minutes

SS – seconds

Fn – n number of fractions

The default format is "YYYY-MM-DD:HH:MI:SS.FFF"

For example:

```
statoprule.timestamp.format=YYYY-MM-DD-HH.MI.SS
```

{oprule}.seqid

Specifies the `seqid` for a particular operation. Use the XPath expression, JMS property, or system value that contains the operation `seqid`. This overrides any `seqid` defined in parent

transaction rules. Must be present if there is no parent transaction rule. This is an optional property.

Any XPath expression must end with a value accessor such as `@att` or `text()`.

```
{oprule}.seqid={xpath-expression}|${jms-property}|*seqid
```

For example:

```
dynoprule.seqid=@seqid
```

{oprule}.txid

Specifies the XPath expression, JMS property, or system value that contains the `txid` used to uniquely identify transactions. This overrides any `txid` defined in parent transaction rules and is required if there is no parent transaction rule. The value must be incremented for each transaction. This is an optional property.

{oprule}.txid={xpath-expression}|${jms-property}|*txid

For most cases using the system value of *txid is preferred.

For example:

dynoprule.txid=$JMSTxId

dynoprule.txid=*txid

{oprule}.txowner

Specifies the XPath expression, JMS property, or static value that contains an arbitrary user name associated with a transaction. This value may be used to exclude certain transactions from processing. This is an optional property.

```
{oprule}.txowner={xpath-expression}|${jms-property}|"{value}"
```

For example:

```
dynoprule.txowner=$MessageOwner
dynoprule.txowner="jsmith"
```

{oprule}.txname

Specifies the XPath expression, JMS property, or static value that contains an arbitrary name to be associated with a transaction. This is an optional property.

```
{oprule}.txname={xpath-expression}|${jms-property}|"value"
```

For example:

```
dynoprule.txname="fixedtx"
```

{oprule}.schemandtable

Specifies the XPath expression JMS property or hard-coded value that contains the schema and table name in the form SCHEMA.TABLE. Any XPath expression must end with a value accessor such as `@att` or `text()`. The value is verified to ensure the table exists in

the source definitions.

```
{oprule}.schemaandtable={xpath-expression}|${jms-property}|"value"
```

For example:

```
statoprule.schemaandtable="MY.TABLE"
```

{oprule}.schema

Specifies the XPath expression, JMS property or hard-coded value that contains the schema name. Any XPath expression must end with a value accessor such as `@att` or `text()`.

{oprule}.schema={xpath-expression}|${jms-property}|"value"

For example:

statoprule.schema=@schema

{oprule}.table

Specifies the XPath expression, JMS property or hard-coded value that contains the table name. Any XPath expression must end with a value accessor such as `@att` or `text()`.

```
{oprule}.table={xpath-expression}|${jms-property}|"value"
```

For example:

```
statoprule.table=$TableName
```

{oprule}.optype

Specifies the XPath expression, JMS property or literal value that contains the `optype` to be validated against an optype `insertval`, etc. Any XPath expression must end with a value accessor such as `@att` or `text()`.

```
{oprule}.optype={xpath-expression}|${jms-property}|"value"
```

For example:

```
dynoprule.optype=@type
statoprule.optype="I"
```

{oprule}.optype.insertval

Specifies the value that identifies an insert operation. The default is I.

```
{oprule}.optype.insertval={value}|\x{hex-value}
```

For example:

```
dynoprule.optype.insertval=A
```

{oprule}.optype.updateval

Specifies the value that identifies an update operation. The default is U.

```
{oprule}.optype.updateval={value}|\x{hex-value}
```

For example:

```
dynoprule.optype.updateval=M
```

{oprule}.optype.deleteval

Specifies the value that identifies a delete operation. The default is D.

```
{oprule}.optype.deleteval={value}|\x{hex-value}
```

For example:

```
dynoprule.optype.deleteval=R
```

{oprule}.txind

Specifies the XPath expression or JMS property that contains the transaction indicator to be validated against `beginval` or other value that identifies the position within the transaction. All operations within a single message are regarded as occurring within a whole transaction if this property is not defined. Specifies the begin, middle and end of transactions. Any XPath expression must end with a value accessor such as `@att` or `text()`. Transactions defined in this way can span messages. This is an optional property.

```
{oprule}.txind={xpath-expression}|${jms-property}
```

For example:

```
dynoprule.txind=@txind
```

{oprule}.txind.beginval

Specifies the value that identifies an operation as the beginning of a transaction. The default is B.

```
{oprule}.txind.beginval={value}|\x{hex-value}
```

For example:

```
dynoprule.txind.beginval=0
```

{oprule}.txind.middleval

Specifies the value that identifies an operation as the middle of a transaction. The default is M.

```
{oprule}.txind.middleval={value}|\x{hex-value}
```

For example:

```
dynoprule.txind.middleval=1
```

{oprule}.txind.endval

Specifies the value that identifies an operation as the end of a transaction. The default is E.

```
{oprule}.txind.endval={value}|\x{hex-value}
```

For example:

```
dynoprule.txind.endval=2
```

{oprule}.txind.wholeval

Specifies the value that identifies an operation as a whole transaction. The default is W.

```
{oprule}.txind.wholeval={value}|\x{hex-value}
```

For example:

```
dynoprule.txind.wholeval=3
```

{colrule}.name

Specifies the XPath expression or hard-coded value that contains a column name. The column index must be specified if this is not and the column name will be resolved from that. If specified the column name will be verified against the source definitions file. Any XPath expression must end with a value accessor such as `@att` or `text()`.

```
{colrule}.name={xpath-expression}|"value"
```

For example:

```
dyncolrule.name=@name
statkeycolrule.name="keycol"
```

{colrule}.index

Specifies the XPath expression or hard-coded value that contains a column index. If not specified then the column name must be specified and the column index will be resolved from that. If specified the column index will be verified against the source definitions file. Any XPath expression must end with a value accessor such as `@att` or `text()`.

```
{colrule}.index={xpath-expression}|"value"
```

For example:

```
dyncolrule.index=@index
statkeycolrule.index=1
```

{colrule}.value

Specifies the XPath expression or hard-coded value that contains a column value. Any XPath expression must end with a value accessor such as `@att` or `text()`. If the XPath expression fails to return any data because a node or attribute does not exist, the column value will be deemed as null. To differentiate between null and missing values (for updates) the `isnull` and `ismissing` properties should be set. The value returned is used for delete before values, and update/insert after values.

```
{colrule}.value={xpath-expression}|"value"
```

For example:

```
statkeycolrule.value=./text()
```

{colrule}.isnull

Specifies the XPath expression used to discover if a column value is null. The XPath expression must end with a value accessor such as `@att` or `text()`. If the XPath expression returns any value, the column value is null. This is an optional property.

```
{colrule}.isnull={xpath-expression}
```

For example:

```
dyncolrule.isnull=@isnull
```

{colrule}.ismissing

Specifies the XPath expression used to discover if a column value is missing. The XPath expression must end with a value accessor such as `@att` or `text()`. If the XPath expression returns any value, then the column value is missing. This is an optional property.

```
{colrule}.ismissing={xpath-expression}
```

For example:

```
dyncolrule.ismissing=./missing
```

{colrule}.before.value

Overrides `{colrule}.value` to specifically say how to obtain before values used for updates or deletes. This has the same format as `{colrule}.value`. This is an optional property.

For example:

```
dyncolrule.before.value=./beforevalue/text()
```

{colrule}.before.isnull

Overrides `{colrule}.isnull` to specifically say how to determine if a before value is null for updates or deletes. This has the same format as `{colrule}.isnull`. This is an optional property.

For example:

```
dyncolrule.before.isnull=./beforevalue/@isnull
```

{colrule}.before.ismissing

Overrides `{colrule}.ismissing` to specifically say how to determine if a before value is missing for updates or deletes. This has the same format as `{colrule}.ismissing`. This is an optional property.

For example:

```
dyncolrule.before.ismissing=./beforevalue/missing
```

{colrule}.after.value

Overrides `{colrule}.value` to specifically say how to obtain after values used for updates or deletes. This has the same format as `{colrule}.value`. This is an optional property.

For example:

```
dyncolrule.after.value=./aftervalue/text()
```

{colrule}.after.isnull

Overrides `{colrule}.isnull` to specifically say how to determine if an after value is null for updates or deletes. This has the same format as `{colrule}.isnull`. This is an optional property.

For example:

```
dyncolrule.after.isnull=./aftervalue/@isnull
```

{colrule}.after.ismissing

Overrides `{colrule}.ismissing` to specifically say how to determine if an after value is missing for updates or deletes. This has the same format as `{colrule}.ismissing`. This is an optional property.

For example:

```
dyncolrule.after.ismissing=./aftervalue/missing
```

CHAPTER 6

UE: Configuring for Message Delivery

After Oracle GoldenGate and a Java are installed, the Oracle GoldenGate adapters can be installed and used. Oracle GoldenGate adapters are available for Windows, Linux and UNIX; but as the shared library is a platform-dependent C library, it must be built specifically for your operating system, hardware architecture, Oracle GoldenGate version and Java runtime version.

Check to see if a build of Oracle GoldenGate for JMS message capture or Java delivery is available for your OS, version of Oracle GoldenGate, and version of Java

Configure the JRE in the user exit properties file

Modify the user exit properties file to point to the location of the Oracle GoldenGate for Java main jar (ggjava.jar) and set any additional JVM runtime boot options as required (these are passed directly to the JVM at startup):

```
javawriter.bootoptions=-Djava.class.path=ggjava/ggjava.jar
-Dlog4j.configuration=log4j.properties -Xmx512m
```

Note the following options in particular:

- java.class.path can include any custom jars in addition to the core application (ggjava.jar). The current directory (.) is included by default in the classpath. You can reference files relative to the Oracle GoldenGate install directory, to allow storing Java property files, Velocity templates and other classpath resources in the dirprm directory. It is also possible to append to the classpath in the Java application properties file.
- The log4j.configuration option specifies a log4j properties file, found in the classpath. There are pre-configured default log4j settings for basic logging (log4j.properties), debug logging (debug-log4j.properties), and detailed trace-level logging (trace-log4j.properties), found in the resources/classes directory.

Once the user exit properties file is correctly configured for your system, it usually remains unchanged. See “User exit properties” on page 73 for additional configuration options.

Configure a data pump to run the user exit

The user exit Extract is configured as a data pump. The data pump consumes a local trail (for example./dirdat/aa) and sends the data to the user exit. The user exit is responsible for processing all the data.

An example of adding a data pump Extract:

```
ADD EXTRACT javaue, EXTTRAILSOURCE ./dirdat/aa
```

The process names and trail names used above can be replaced with any valid name: process names must be 8 characters or less, trail names must to be two characters. In the user exit Extract parameter file (javaue.prm) specify the location of the user exit library:

Table 2 User exit Extract parameters

Parameter	Explanation
`EXTRACT javaue`	All Extract parameter files start with the Extract name
`SOURCEDEFS ./dirdef/tcust.def`	The Extract process requires metadata describing the trail data. This can come from a database or a source definitions file. This metadata defines the column names and data types in the trail being read (./dirdat/aa).
`SETENV (GGS_USEREXIT_CONF = "dirprm/javaue.properties")`	(Optional) An absolute or relative path (relative to the Extract executable) to the properties file for the C user exit library. The default value is javawriter.properties in the same directory as Extract.
`SETENV (GGS_JAVAUSEREXIT_CONF = "dirprm/javaue.properties")`	(Optional) The Java properties.This example places the properties file in the dirpm directory.
`CUSEREXIT javaue/Java6_UserExit.dll CUSEREXIT PASSTHRU INCLUDEUPDATEBEFORES`	The CUSEREXIT parameter includes the following: ◆ The location of the user exit library. For UNIX, the library would be suffixed `.so` ◆ CUSEREXIT - the callback function name that must be uppercase. ◆ PASSTHRU - avoids the need for a dummy target trail. ◆ INCLUDEUPDATEBEFORES - needed for transaction integrity.
`TABLE schema.*;`	The tables to pass to the User Exit; tables not included will be skipped. *No filtering* may be done in the user exit Extract; otherwise transaction markers will be missed. You can filter in the primary Extract, or use another, upstream data pump, or filter data directly in the Java application.

The two environment properties show above are optional, but useful. For example, these allow you to place all your properties files in the dirprm directory instead of the default locations:

- `SETENV (GGS_USEREXIT_CONF = "dirprm/javaue.properties")`

  This changes the default configuration file used for the User Exit shared library. The value given is either an absolute path, or a path relative to Extract (or Replicat). The default file used is javawriter.properties, located in the same directory as Extract. The example above uses a relative path to put this property file in the dirprm directory.

- `SETENV (GGS_JAVAUSEREXIT_CONF = "dirprm/javaue.properties")`

  This changes the default properties file used for the Oracle GoldenGate for Java framework. The value found is a path to a file found in the classpath or in a normal file system path.

Configure the Java handlers

The Oracle GoldenGate Java API has a property file used to configure active event handlers. To test the configuration, you may use the built-in file handler or the logger, stderr or stdout handlers (typically used for testing and debugging only). Here are some example properties, followed by explanations of the properties (comment lines start with #):

```
# the list of active handlers
gg.handlerlist=myhandler
# set properties on 'myhandler'
gg.handler.myhandler.type=file
gg.handler.myhandler.format=com/goldengate/atg/datasource/tx2xml.vm
gg.handler.myhandler.file=output.xml
```

This property file declares the following:

- A single event handler is active, called myhandler. Multiple handlers may be specified, separated by commas. For example: gg.handlerlist=myhandler, yourhandler
- Configuration of the handlers. For example, to set the myhandler property “color” to “blue”: gg.handler.myhandler.color=blue

  NOTE See the documentation for each type of handler (e.g. the JMS handler or the File writer handler) for the list of valid properties that may be set.

- The format of the output is defined by the Velocity template tx2xml.vm. In this case, this file is included inside the application jar file. You may also specify your own custom template to define the message format; just specify the path to your template relative to the Java classpath (this is discussed later).

This property file is actually a complete example that will write captured transactions to the output file output.xml. Other handler types can be specified using the key words: **jms_text** (or **jms**), **jms_map**, **stdout**, **stderr**, **logger** (logs to log4j), **singlefile** (a file that does not roll), and others. Custom handlers can be implemented, in which case the type would be the fully qualified name of the Java class for the handler.

CHAPTER 7
UE: Running the User Exit

This section assumes that the primary Extract has already generated a trail to be consumed by the user exit Extract.

Starting the application

To run the user exit and execute the Java application, you only need an existing trail file and its corresponding sourcedef file. For the examples that follow, a simple TCUSTMER and TCUSTORD trail is used (matching the demo SQL provided with the Oracle GoldenGate software download), along with a source definitions file defining the data types used in the trail.

NOTE The user exit does not require access to a database in order to run. But the Extract process does require metadata describing the trail data. Either the Extract must login to a database for metadata, or a source definitions file can be provided. In either case, the Extract cannot be in PASSTHRU mode when using a user exit.

To run the user exit, simply start the Extract process from GGSCI:

```
GGSCI> START EXTRACT javaue
GGSCI> INFO EXTRACT javaue
EXTRACT JAVAUE Last Started 2008-03-25 18:41 Status RUNNING
Checkpoint Lag 00:00:00 (updated 00:00:00 ago)
Log Read Checkpoint File ./dirdat/bb000000
2007-09-24 12:52:58.000000 RBA 2702
```

If the Extract process is running and the file handler is being used (as in the example above), then you should see the output file output.xml in the Oracle GoldenGate installation directory (the same directory as the Extract executable).

If the process does not start or abends, see "Error handling" on page 92:

Restarting the application at the beginning of a trail

There are two checkpoints for an Extract running the user exit: the user exit checkpoint and the Extract checkpoint. Before rerunning the Extract, you must reset both checkpoints:

1. Delete the user exit checkpoint file.

   The sample properties file has `goldengate.userexit.chkptprefix=JAVAUE_` in the user exit properties file.

Windows: `cmd> del JAVAUE_javawriter.chkpt`

UNIX: `$ rm JAVAUE_javawriter.chkpt`

> **NOTE** Do not modify checkpoints or delete the user exit checkpoint file on a production system.

2. Reset the Extract to the beginning of the trail data:

```
GGSCI> ALTER EXTRACT JAVAUE, EXTSEQNO 0, EXTRBA 0
```

3. Restart the Extract:

```
GGSCI> START JAVAUE
GGSCI> INFO JAVAUE
EXTRACT    JAVAUE    Last Started 2008-03-25 18:41   Status RUNNING
Checkpoint Lag       00:00:00 (updated 00:00:00 ago)
Log Read Checkpoint  File ./dirdat/ps000000
                     2007-09-24 12:52:58.000000  RBA 2702
```

It may take a couple seconds for the Extract process status to report itself as running. Check the report file to see if it abended or is still in the process of starting:

```
GGSCI> VIEW REPORT JAVAUE
```

CHAPTER 8
UE: Configuring Event Handlers

Specifying event handlers

Processing transaction, operation and metadata events in Java works as follows:

- The Oracle GoldenGate Extract reads local trail data and passes the transactions, operations and database metadata to the user exit. Metadata can come from either a source definitions file or by querying the database.
- Events are fired by the Java framework, optionally filtered by custom Event Filters.
- Handlers (event listeners) process these events, and process the transactions, operations and metadata. Custom formatters may be applied for certain types of targets.

There are several existing handlers:

- Sending messages to a JMS provider using either a MapMessage, or using a TextMessage with customizable formatters.
- Simple handlers for logging to log4j and to stdout/stderr.
- A filewriter handler, for writing to a single file, or a rolling file.

NOTE The filewriter handler is particularly useful as development utility, since the filewriter can take the exact same formatter as the JMS TextMessage handler. Using the filewriter provides a simple way to test and tune the formatters for JMS without actually sending the messages to JMS.

Event handlers can be configured using the main Java property file or they may optionally read in their own properties directly from yet another property file (depending on the handler implementation). Handler properties are set using the following syntax:

```
gg.handler.{id}.someproperty=somevalue
```

This will cause the property *someproperty* to be set to the value *somevalue* for the handler instance identified in the property file by {id}. This {id} is used only in the property file to define active handlers and set their properties; it is user-defined, and has no meaning outside of the property file.

Implementation note (for Java developers): Following the above example: when the handler is instantiated, the method `void` setSomeProperty(String value) will be called on the handler instance, passing in the String value *somevalue*. A JavaBean PropertyEditor may also be defined for the handler, in which case the string can be automatically converted to the appropriate type for the setter method. For example, in the Java application properties file, we may have the following (comment lines start with #):

```
# the list of active handlers: only two are active
gg.handlerlist=one, two

# set properties on 'one'
gg.handler.one.type=file
gg.handler.one.format=com.mycompany.MyFormatter
gg.handler.one.file=output.xml

# properties for handler 'two'
gg.handler.two.type=jms_text
gg.handler.two.format=com.mycompany.MyFormatter
gg.handler.two.properties=jboss.properties
# set properties for handler 'foo'; this handler is ignored
gg.handler.foo.type=com.mycompany.MyHandler
gg.handler.foo.someproperty=somevalue
```

The type identifies the handler class; the other properties depend on the type of handler created. If a separate properties file is used to initialize the handler (such as the JMS handlers), the properties file is found in the classpath. For example, if properties file is at: `{gg_install_dir}/dirprm/foo.properties`, then specify in the properties file as follows: `gg.handler.{id}.properties=foo.properties`.

JMS handler

The main Java property file identifies active handlers. The JMS handler may optionally use a separate property file for JMS-specific configuration. This allows more than one JMS handler to be configured to run at the same time.

There are examples included for several JMS providers (JBoss, TIBCO, Solace, ActiveMQ, WebLogic). For a specific JMS provider, you can choose the appropriate properties files as a starting point for your environment. Each JMS provider has slightly different settings, and your environment will have unique settings as well.

The installation directory for the Java jars (ggjava) contains the core application jars (ggjava.jar) and its dependencies in `resources/lib/*.jar`. The resources directory contains all dependencies and configuration, and is in the classpath:.

If the JMS client jars already exist somewhere on the system, they can be referenced directly and added to the classpath without copying them.

The following properties are typically set for a JMS handler (which is a JMS producer, publishing messages to a JMS queue or topic). The example illustrates using JBoss messaging.

Table 3 JMS property settings

Property	Description
`gg.jmshandler.persistent=false`	If the messages are to be persistent, the JMS provider must be configured to log the message to stable storage as part of the client's send operation.

Table 3 JMS property settings

Property	Description
`gg.jmshandler.queueortopic=queue` `gg.jmshandler.destination=queue/A`	Can be set to queue or topic. The destination name for the queue or topic can be configured in the JMS provider; this needs to be provided by the JMS administrator.
`gg.jmshandler.user=myusername` `gg.jmshandler.password=mypassword`	If authentication is required by the JMS provider.
`gg.jmshandler.connectionfactory=ConnectionFactory`	JNDI connection factory name to lookup.

`java.naming.provider.url=localhost:1099`
`java.naming.factory.url.pkgs=`
`jboss.naming:org.jnp.interfaces`
`java.naming.factory.initial=`
`org.jnp.interfaces.NamingContextFactory`
	Standard JNDI properties for InitialContext (each property set on a single line, without wrapping). The key property is the host and port of the JNDI.

Here is the main properties file, specifying the additional JMS property file:

```
# one JMS handler active, using Velocity template formatting
gg.handlerlist=myjms
gg.handler.myjms.type=jms_text
gg.handler.myjms.format=/templates/sample2xml.vm
gg.handler.myjms.properties=/dirprm/jboss.properties
gg.handler.myjms.classpath=/usr/jboss/client/*, /path/to/my/app.jar
```

There are two types of JMS handlers which may be specified:

- **jms_text** – sends text messages to a topic or queue. The messages may be formatted using Velocity templates or by writing a formatter in Java. The same formatters can be used for a jms_text message as for writing to files. (**jms** is a synonym for jms_text.)
- **jms_map** – sends a JMS MapMessage to a topic or queue. The JMSType of the message is set to the name of the table. The body of the message consists of the following metadata, followed by column name and column value pairs:
  - GG_ID – position of the record, uniquely identifies this operation
  - GG_OPTYPE – type of SQL (insert/update/delete),
  - GG_TABLE – table name on which the operation occurred
  - GG_TIMESTAMP – timestamp of the operation

File handler

The file handler is often used to verify the message format when the actual target is JMS, and the message format is being developed using custom Java or Velocity templates. Here

is a property file using a file handler:

```
# one file handler active, using velocity template formatting
gg.handlerlist=myfile
gg.handler.myfile.type=file
gg.handler.myfile.rollover.size=5M
gg.handler.myfile.format=/dirprm/sample2xml.vm
gg.handler.myfile.file=output.xml
```

This example uses a single handler (though, a JMS handler and the file handler could be used at the same time), writing to a file called output.xml, using a velocity template called sample2xml.vm. The template is found via the classpath.

Custom handlers

For information on coding a custom handler, see “Coding a custom handler in Java” on page 88.

Formatting the output

As previously described, the existing JMS and file output handlers can be configured through the properties file. Each handler has its own specific properties that can be set: for example, the output file can be set for the file handler, and the JMS destination can be set for the JMS handler. Both of these handlers may also specify a custom formatter. The same formatter may be used for both handlers. As an alternative to writing Java code for custom formatting, a Velocity template may be specified. For further information, see “Custom formatting” on page 85.

Reporting

Summary statistics about the throughput and amount of data processed are generated when the Extract process stops. Additionally, statistics can be written periodically after a specified amount of time or after a specified number of records have been processed. If both time and number of records are specified, then the report is generated for whichever event happens first. These statistical summaries are written to the Oracle GoldenGate report file and the user exit log files.

CHAPTER 9
UE: Message Delivery Properties

This section explains the options available for configuration of the property files for:

- **User exit properties**
- **Java application properties**

Place these two property files in the `dirprm` directory of your Oracle GoldenGate installation location. Names for the user exit and Java application properties file are set through environmental variables.

```
SETENV (GGS_USEREXIT_CONF = "dirprm/javaue.properties")
SETENV (GGS_JAVAUSEREXIT_CONF = "dirprm/javaue.properties")
```

All properties in the property file are of the form: `fully.qualified.name=value`. The value may be integer, boolean, single string, or comma delimited strings.

Comments can be entered in to the properties file with the # prefix at the beginning of the line. For example:

```
# This is a property comment
some.property=value
```

Properties themselves can also be commented out. However you cannot place a comment at the end of a line; either the whole line is a comment or it is a property.

User exit properties

Logging properties

Logging is controlled by the following properties.

log.logname

Specifies the prefix to the log file name. This must be a valid ASCII string. The log file name has the current date appended to it, in yyyymmdd format, together with the .log extension.

The following example will create a log file of name writer_20100803.log on August 3, 2010.

The log file will roll over each day, independent of the stopping and starting of the process.

```
# log file prefix
log.logname=writer
```

The following example will create a log file of name msgv_20100803.log on August 3, 2010.

```
# log file prefix
log.logname=msgv
```

log.level

Specifies the overall log level for all modules. The syntax is:

```
log.level=ERROR|WARN|INFO|DEBUG
```

The log levels are defined as follows:

ERROR – Only write messages if errors occur

WARN – Write error and warning messages

INFO – Write error, warning and informational messages

DEBUG – Write all messages, including debug ones.

The default logging level is INFO. The messages in this case will be produced on startup, shutdown and periodically during operation. If the level is switched to DEBUG, large volumes of messages may occur which could impact performance. For example, the following sets the global logging level to INFO:

```
# global logging level
log.level=INFO
```

log.tostdout

Controls whether or not log information is written to standard out. This setting is useful if the Extract process is running with a VAM started from the command line or on an operating system where `stdout` is piped into the report file. Generally Oracle GoldenGate processes run as background processes however.

The syntax is:

```
goldengate.log.tostdout=true|false
```

The default is false.

log.tofile

Controls whether or not log information is written to the specified log file.The syntax is:

```
log.tofile=true|false
```

The default is false. Log output is written to the specified log file when set to true.

log.modules, log.level.{module}

Specifies the log level of the individual source modules that comprise the user exit. This is typically used only for advanced debugging. It is possible to increase the logging level to DEBUG on a per module basis to help troubleshoot issues. The default levels should not be changed unless asked to do so by Oracle GoldenGate support.

General properties

The following properties apply to all writer type user exits and are not specific to the user exit.

goldengate.userexit.writers

Specifies the name of the writer. This is always `javawriter` and should not be modified.

For example:

```
goldengate.userexit.writers=javawriter
```

All other properties in the file should be prefixed by the writer name, javawriter.

goldengate.userexit.chkptprefix

Specifies a string value for the prefix added to the checkpoint file name. For example:

```
goldengate.userexit.chkptprefix=javaue_
```

Should the Extract process have to be repositioned back in the trail (for example, to the beginning of the trail), then this checkpoint file will have to be deleted.

JVM boot options

The following options configure the Java Runtime Environment. In particular, this is where the maximum memory the JVM will use can be specified; if you see Java out-of-memory errors, edit these settings.

javawriter.bootoptions

Specifies the classpath and boot options that will be applied when the user exit starts up the JVM. The path needs colon (:) separators for UNIX/Linux and semicolons (;) for Windows. This is where to specify various options for the JVM, such as heap size and classpath; for example:

-Xms: initial java heap size

-Xmx: maximum java heap size

-**Djava.class.path**: classpath specifying location of at least the main application jar, ggjava.jar. Other jars, such as JMS provider jars, may also be specified here as well; alternatively, these may be specified in the Java application properties file.

-verbose:jni: run in verbose mode (for JNI)

For example (all on a single line):

```
javawriter.bootoptions= -Djava.class.path=ggjava/ggjava.jar
-Dlog4j.configuration=my-log4j.properties -Xmx512m
```

The log4j.configuration property could be a fully qualified URL to a log4j properties file; by default this file is searched for in the classpath. You may use your own log4j configuration, or one of the pre-configured log4j settings: log4j.properties (default level of logging), debug_log4j.properties (debug logging) or trace_log4j.properties (very verbose logging).

Statistics and reporting

The use of the user exit causes Extract to assume that the records handled by the exit are ignored. This causes the standard Oracle GoldenGate reporting to be incomplete. Oracle GoldenGate for Java adds its own reporting to handle this issue.

Statistics can be reported every `t` seconds or every `n` records - or if both are specified, whichever criteria is met first.

There are two sets of statistics recorded: those maintained by the User Exit shared library (on the C side) and those obtained from the Java libraries. The reports received from the Java side are formatted and returned by the individual handlers.

The User Exit statistics include the total number of operations, transactions and corresponding rates.

javawriter.stats.display

Controls the output of statistics to the Oracle GoldenGate report file and to the user exit log files.

The following example outputs these statistics.

```
javawriter.stats.display=true
```

javawriter.stats.full

Controls the output of statistics from the Java side, in addition to the statistics from the C side.

Java side statistics are more detailed but also involve some additional overhead, so if statistics are reported often and a less detailed summary is adequate, it is recommended that `stats.full` property is set to false.

The following example will output Java statistics in addition to C.

```
javawriter.stats.full=true
```

javawriter.stats.{time, numrecs}

Specifies a time interval in seconds or a number of records, after which statistics will be reported. The default is to report statistics every hour or every 10000 records (which ever occurs first).

For example, to report ever 10 minutes or every 1000 records, specify:

```
javawriter.stats.time=600
javawriter.stats.numrecs=1000
```

The Java application statistics are handler-dependent:

- For the all handlers, there is at least the total elapsed time, processing time, number of operations, transactions;
- For the JMS handler, there is additionally the total number of bytes received and sent.
- The report can be customized using a template.

Java application properties

The following defines the properties which may be set in the Java application property file.

Properties for all handlers

The following properties apply to all handlers.

gg.handlerlist

The handler list is a comma-separated list of active handlers. These values are used in the rest of the property file to configure the individual handlers. For example:

```
gg.handlerlist=name1, name2
gg.handler.name1.propertyA=value1
gg.handler.name1.propertyB=value2
gg.handler.name1.propertyC=value3
gg.handler.name2.propertyA=value1
gg.handler.name2.propertyB=value2
gg.handler.name2.propertyC=value3
```

Using the `handlerlist` property, you may include completely configured handlers in the property file and just disable them by removing them from the handlerlist.

gg.handler.{name}.type

The type of handler is either a predefined value for built-in handlers, or a fully qualified Java class name. The syntax is:

```
gg.handler.{name}.type={jms | jms_map | singlefile | rolling | stdout |
stderr | log | com.foo.MyHandler}
```

Where: All but the last are pre-defined handlers:

jms: sends transactions/operations/metadata as formatted messages to a JMS provider

jms_map: sends JMS map messages

singlefile: writes to a single file on disk, but does not roll the file

rolling: writes transactions/operations/metadata to a file on disk, rolling the file over after a certain size or after a certain amount of time

stdout, stderr, log: write to stdout, stderr, and log4j outputs, respectively. stdout and stderr accept a formatter; log4j just prints a summary of the transaction/operation

custom Java class: any class extending the Oracle GoldenGate for Java AbstractHandler class may handle transaction/operation/metadata events.

Properties for formatted output

The following properties apply to all handlers capable of producing formatted output; this includes:

- The jms_text handler (but not the jms_map handler)
- The singlefile and rolling handlers, for writing formatted output to files
- The stdout and stderr handlers, for generating output directly to the console.

gg.handler.{name}.format

Specifies the format used to transform operations and transactions into messages sent to JMS or to a file. The format is specified uniquely for each handler. The value may be:

- **Velocity template**
- **Java class name** (fully qualified - the class specified must be a type of formatter)
- **csv** for delimited values (such as comma-separated values; the delimiter can be customized)
- **fixed** for fixed-length fields
- **Built-in formatter**, such as:

  `xml` – demo XML format (this format may change in future releases)

  `xml2` – internal XML format (this format may change in future releases)

For example, to specify a custom Java class:

```
gg.handlerlist=abc
gg.handler.abc.format=com.mycompany.MyFormat
```

Or, for a Velocity template:

```
gg.handlerlist=xyz
gg.handler.xyz.format=path/to/sample.vm
```

If using templates, the file is found relative to some directory or jar that is in the classpath. By default, the Oracle GoldenGate install directory is in the classpath, so the above template could be placed in the `/dirprm` directory of the Oracle GoldenGate installation location.

The default format is to use the built-in XML formatter.

gg.handler.{name}.includeTables

Specifies a list of tables to include by this handler. If the schema (or owner) of the table is specified, then only that schema matches the table name; otherwise, the table name matches any schema. A list of tables may be specified, commaseparated.

For example, to have the handler only process tables foo.customer and bar.orders:

```
gg.handler.myhandler.includeTables=foo.customer, bar.orders
```

NOTE In order to selectively process operations on a table by table basis, the handler must be processing in operation mode. If the handler is processing in transaction mode, then when a single transaction contains several operations spanning several tables, if any table matches the include list of tables, the transaction will be included.

gg.handler.{name}.excludeTables

Specifies a list of tables to exclude by this handler. If the schema (or owner) of the table is specified, then only that schema matches the table name; otherwise, the table name matches any schema. A list of tables may be specified, comma-separated. For example, to have the handler process all operations on all tables *except* table `date_modified` in all schemas:

```
gg.handler.myhandler.excludeTables=date_modified
```

gg.handler.{name}.mode, gg.handler .{name}.format.mode

Specifies whether to output one operation per message (`op`) or one transaction per message (`tx`). The default is `op`. Use `format.mode` when you have a custom formatter.

Properties for CSV and fixed-format output

If the handler is set to use either CSV or fixed-format output, the following properties may also be set. Many of the same properties apply for both formats; there is however no unique prefix to the property settings. If there is more than one handler requiring unique settings, these properties can be set in a separate properties file. For example, if there are two JMS handlers, each using CSV or fixed-format:

```
gg.handler.my_jms_handler1.type=jms_text
gg.handler.my_jms_handler1.format=csv
gg.handler.my_jms_handler1.properties=/dirprm/my-csv.properties
...
gg.handler.my_jms_handler2.type=jms_text
gg.handler.my_jms_handler2.format=fixed
gg.handler.my_jms_handler2.properties=/dirprm/my-fixed.properties
...
```

delim

Specifies the delimiter to use between fields (set to no value in order to have no delimiter used). For example:

```
delim=,
```

quote

Specifies the quote character to be used if column values are quoted. For example:

```
quote='
```

metacols

Specifies the metadata column values to appear at the beginning of the record, before any column data. Specify any of the following, in the order they should appear:

- **position:** unique position indicator of records in a trail
- **opcode:** `I`, `U`, or `D` for insert/update/delete records (see: `insertChar`, `updateChar`, `deleteChar`)
- **txind:** transaction indicator – e.g. 0=begin, 1=middle, 2=end (3=whole tx) (see `beginTxChar`, `middleTxChar`, `endTxChar`, `wholeTxChar`)
- **opcount:** position of a record in a transaction, starting from 0
- **schema:** schema/owner of the table for the record
- **tableonly:** just table (no schema/owner)
- **table:** full name of table, schame.table
- **timestamp:** commit timestamp of record

For example:

```
metacols=opcode, table, txind, position
```

missingColumnChar, presentColumnChar, nullColumnChar

Specifies a special column prefix for columns values that are:

- **present**: column value exists in the trail and is non-NULL
- **missing**: column value not in trail; it is unknown if it has a value or is NULL. It was not captured from the source database transaction log.
- **null**: column value is set to NULL

The character used to represent these special states can be customized (by default, they are set to empty string and do not show). For example:

```
missingColumnChar=M
presentColumnChar=P
nullColumnChar=N
```

beginTxChar, middleTxChar, endTxChar, wholeTxChar

Specifies the header metadata characters (see metacols): used to identify a record as the begin, middle, or end of a transaction. If one operation consists of a complete Tx, then it's a "whole" transaction. For example:

```
beginTxChar=B
middleTxChar=M
endTxChar=E
wholeTxChar=W
```

insertChar, updateChar, deleteChar

Specifies the characters to identify insert/update/delete (by default, `I`/`U`/`D`). For example, to use `INS`, `UPD`, and `DEL` instead of `I`, `U` and `D` for insert, update, and delete operations:

```
insertChar=INS
updateChar=UPD
deleteChar=DEL
```

endOfLine

Specifies the end-of-line character as:

- Native platform: `EOL`
- Neutral (UNIX-style \n): `CR`
- Windows (\r\n): `CRLF`

For example:

```
endOfLine=CR
```

justify

Specifies the left or right justification of fixed fields. For example:

```
justify=left
```

includeBefores

Controls whether before images should be included in the output. There must be before images in the trail. For example:

```
includeBefores=false
```

File writer properties

The following properties only apply to handlers that write their output to files: the file handler and the singlefile handler.

gg.handler.{name}.file

Specifies the name of the output file for the given handler. If the handler is a rolling file, this filename is used to derive the rolled filenames. The default filename is output.xml.

gg.handler.{name}.append

Controls whether the file should be appended to (true) or overwritten upon restart (false).

gg.handler.{name}.rolloverSize

If using the file handler, this specifies the size of the file before a rollover should be attempted. The file size will be at least this size, but will most likely be larger. Operations and transactions are not broken across files. The size is specified in bytes, but a suffix may be given to identify MB, or KB. For example:

```
gg.handler.myfile.rolloverSize=5M
```

The default rollover size is 10 MB.

JMS handler properties

The following properties apply to the JMS handlers. Some of these values may be defined in the Java application properties file using the name of the handler. Other properties may be placed into a separate JMS properties file, which is useful if using more than one JMS handler at a time. For example:

```
gg.handler.myjms.type=jms_text
gg.handler.myjms.format=xml
gg.handler.myjms.properties=/dirprm/jboss.properties
```

Just as with Velocity templates and formatting property files, this additional JMS properties file is found in the classpath. The above properties file jboss.properties would be found in `{gg_install_dir}/dirprm/jboss.properties`, since the `dirprm` directory is included by default in the classpath.

Settings that can be made in the Java application properties file will override the corresponding value set in the supplemental JMS properties file (`jboss.properties` in the example above). In the following example, the destination property is specified in the Java application properties file. This allows the same default connection information for the two handlers `myjms1` and `myjms2`, but customizes the target destination queue:

```
gg.handler.myjms1.type=jms_text
gg.handler.myjms1.destination=queue.sampleA
gg.handler.myjms1.format=/dirprm/sample.vm
gg.handler.myjms1.properties=/dirprm/tibco-default.properties
gg.handler.myjms2.type=jms_map
gg.handler.myjms2.destination=queue.sampleB
gg.handler.myjms2.properties=/dirprm/tibco-default.properties
```

The following properties can be specified in the JMS handler itself; see further below for all other properties that can be specified in the generic JMS properties file.

NOTE The property names that can be used in the Java application properties file are similar, but not identical to the properties specified in the generic JMS properties file. In particular, watch out for additional periods (.) in the property names. Also, not all properties that may be specified in the generic JMSproperty file can be overridden by the handler specification in the Java application properties file.

The property names are listed here with brief descriptions. For further details, see "Standard JMS settings" on page 83.

destination: queue or topic name

queueOrTopic: [queue | topic]

user : user name for queue/topic, if required

password: password for queue/topic, if required

persistent: [true | false]

priority: integer

timetolive: milliseconds

connectionUrl: (if useJndi=false) URL to connect to queue/topic

connectionFactoryClass: (if useJndi=false) class name to instantiate

useJndi: [true | false]

connectionFactory: (if useJndi=true) connection factory name to look up

To use any of the above settings, specify the handler name as a prefix; for example:

```
gg.handlerlist=sample,sample2
gg.handler.sample.type=jms_text
gg.handler.sample.format=/dirprm/my_template.vm
gg.handler.sample.destination=gg.myqueue
gg.handler.sample.queueortopic=queue
gg.handler.sample.connectionUrl=tcp://host:61616?jms.useAsyncSend=true
gg.handler.sample.properties=activemq-default.properties
gg.handler.sample2.type=jms_map
gg.handler.sample2.destination=gg.mytopic
gg.handler.sample2.queueortopic=topic
gg.handler.sample2.connectionUrl=tcp://host2:61616?jms.useAsyncSend=true
gg.handler.sample2.properties=activemq-default.properties
```

And the corresponding additional JMS properties file (`activemq-default.properties`) might contain the following values, some of which provide default values for both handlers sample and sample2 (`timetolive`, `sessionmode`), and others are unused (`destination=ggdemo.queueA`, connection URL to `localhost`, the `queueortopic` setting):

```
gg.jmshandler.queueortopic=queue
gg.jmshandler.destination=ggdemo.queueA
gg.jmshandler.sessionmode=dupsok
gg.jmshandler.durabletopic=false
gg.jmshandler.usejndi=false
gg.jmshandler.connectionfactory=ConnectionFactory
gg.jmshandler.connection.factoryclass=\
org.apache.activemq.ActiveMQConnectionFactory
gg.jmshandler.connection.url=tcp://localhost:61616?jms.useAsyncSend=true
gg.jmshandler.timetolive=50000
```

Standard JMS settings

The following outlines the JMS properties which may be set, and the accepted values. These apply for both JMS handler types: jms_text (TextMessage) and jms_map (MapMessage).

JMS Destination: destination - This the queue or topic to which the message is sent. This must be correctly configured on the JMS server. Typical values may be: queue/A, queue.Test, example.MyTopic, etc.

JMS User: user – user name required to send messages to the JMS server (optional)

JMS Password: password – password required to send messages to the JMS server (optional)

JMS Queue or Topic: queueortopic=[queue, topic] – whether the handler is sending to a queue (a single receiver) or a topic (publish / subscribe). This must be correctly configured in the JMS provider.

queue: a message is removed from the queue once it has been read.

topic: messages are published and may be delivered to multiple subscribers.

JMS Persistent: persistent=[true, false] – if the delivery mode is set to persistent or not.

If the messages are to be persistent, the JMS provider must be configured to log the message to stable storage as part of the client's send operation.

JMS Priority: priority – JMS defines a 10 level priority value, with 0 as the lowest and 9 as the highest. Clients should consider 04 as gradients of normal priority and 59 as gradients of expedited priority. Priority is set to 4, by default.

JMS TimeToLive: timetolive – the default length of time in milliseconds from its dispatch time that a produced message should be retained by the message system. Time to live is set to zero by default (zero is unlimited).

JMS Connection Factory Name: .connectionfactory – name of the connection factory to lookup via JNDI

JMS Use JNDI: usejndi=[true, false] – if usejndi is false, then JNDI is not used to configure the JMS client. Instead, factories and connections are explicitly constructed.

JMS Url: connection.url – connectionUrl used only when `usejndi=false`, to explicitly create the connection.

JMS ConnectionFactoryClass: connection.factoryclass – connectionFactoryClass only used if `usejndi=false`. When not relying on JNDI to access a factory, the value of this property is the Java classname to instantiate, constructing a factory object explicitly.

gg.handlerlist.nop

In addition to the other JMS properties, there is a debug "nop" property that can be globally set which disables the sending of the JMS messages altogether. This is only used for testing purposes. The events are still generated and handled and the message is constructed – it is simply not sent. This can be used to test the performance of the message generation. It can be set to true or false (the default is false). For example:

```
gg.handlerlist.nop=true
```

General properties

The following are general properties that are used for the user exit Java framework.

gg.classpath

Specifies additional directories or jars to add to classpath.

gg.report.format

Specifies the template to use for customizing the report format.

CHAPTER 10
UE: Developing Custom Filters, Formatters and Handlers

You can write Java code to implement an event filter, a custom formatter for a built-in handler or a custom event handler. You can also specify custom formatting through a Velocity template.

Filtering events

By default, all transactions, operations and metadata events are passed to the `DataSourceListener` event handlers. In order to filter which events are actually sent to the handlers (e.g. to only process certain operations on certain tables containing certain column values), an event filter can be implemented.

Filters are additive: if more than one filter is set for a handler, then all filters must return true in order for the event to be passed to the handler.

You can configure filters using the Java application properties file:

```
# handler "foo" only receives certain events
gg.handler.one.type=jms
gg.handler.one.format=/dirprm/mytemplate.vm
gg.handler.one.filter=com.mycompany.MyFilter
```

That is, all you have to do is write the filter, set it on the handler, and then the filter is active: no additional logic needs to be added to specific handlers.

Custom formatting

You can customize the output format of a built-in handler by:

- Writing a custom formatter in Java or
- Using a Velocity template

Coding a custom formatter in Java

The earlier examples shows a JMS handler and a file output handler using the same formatter (com.mycompany.MyFormatter); following is an example of how this formatter

may be implemented:

```
package com.mycompany.MyFormatter;

import com.goldengate.atg.datasource.DsOperation;
import com.goldengate.atg.datasource.DsTransaction;
import com.goldengate.atg.datasource.format.DsFormatterAdapter;
import com.goldengate.atg.datasource.meta.ColumnMetaData;
import com.goldengate.atg.datasource.meta.DsMetaData;
import com.goldengate.atg.datasource.meta.TableMetaData;
import java.io.PrintWriter;

public class MyFormatter extends DsFormatterAdapter {
    public MyFormatter() { }

    @Override
    public void formatTx(DsTransaction tx,
              DsMetaData meta,
              PrintWriter out)
    {
        out.print("Transaction: " );
        out.print("numOps=\'" + tx.getSize() + "\' " );
        out.println("ts=\'" + tx.getStartTxTimeAsString() + "\'");

        for(DsOperation op: tx.getOperations()) {
            TableName currTable = op.getTableName();
            TableMetaData tMeta = dbMeta.getTableMetaData(currTable);
            String opType = op.getOperationType().toString();
            String table = tMeta.getTableName().getFullName();
            out.println(opType + " on table \"" + table + "\":" );
            int colNum = 0;
            for(DsColumn col: op.getColumns())
            {
                ColumnMetaData cMeta = tMeta.getColumnMetaData( colNum++ );
                out.println(
                cMeta.getColumnName() + " = " + col.getAfterValue() );
            }
    }

    @Override
    public void formatOp(DsTransaction tx,
              DsOperation op,
              TableMetaData tMeta,
              PrintWriter out)
    {
        // not used...
    }
}
```

The formatter defines methods for either formatting complete transactions (after they are committed) or individual operations (as they are received, before the commit). If the formatter is in operation mode, then `formatOp`(...) is called; otherwise, `formatTx`(...) is called at transaction commit.

To compile and use this custom formatter, include the Oracle GoldenGate for Java jars in the classpath and place the compiled .class files in {gg_install_dir}/dirprm:

```
javac -d {gg_install_dir}/dirprm
-classpath ggjava/ggjava.jar MyFormatter.java
```

The resulting class files would be located in resources/classes (in correct package structure):

```
{gg_install_dir}/dirprm/com/mycompany/MyFormatter.class
```

Alternatively, the custom classes can be put into a jar; in this case, either include the jar file in the JVM classpath via the user exit properties (using java.class.path in the javawriter.bootoptions property), or by setting the Java application properties file to include your custom jar:

```
# set properties on 'one'
gg.handler.one.type=file
gg.handler.one.format=com.mycompany.MyFormatter
gg.handler.one.file=output.xml
gg.classpath=/path/to/my.jar,/path/to/directory/of/jars/*
```

Using a Velocity template

As an alternative to writing Java code for custom formatting, Velocity templates can be a good alternative to quickly prototype formatters. For example, the following template could be specified as the format of a JMS or file handler:

```
Transaction: numOps='$tx.size' ts='$tx.timestamp'
#for each( $op in $tx )
operation: $op.sqlType, on table "$op.tableName":
#for each( $col in $op )
$op.tableName, $col.meta.columnName = $col.value
#end
#end
```

If the template were named sample.vm, it could be placed in the classpath, for example:

```
{gg_install_dir}/dirprm/sample.vm
```

NOTE If using Velocity templates, the filename must end with the suffix .vm; otherwise the formatter is presumed to be a Java class.

Update the Java application properties file to use the template:

```
# set properties on 'one'
gg.handler.one.type=file
gg.handler.one.format=/dirprm/sample.vm
gg.handler.one.file=output.xml
```

When modifying templates, there is no need to recompile any Java source; simply save the template and re-run the Java application. When the application is run, the following output would be generated (assuming a table named SCHEMA.SOMETABLE, with columns TESTCOLA and TESTCOLB):

```
Transaction: numOps='3' ts='2008-12-31 12:34:56.000'
operation: UPDATE, on table "SCHEMA.SOMETABLE":
SCHEMA.SOMETABLE, TESTCOLA = value 123
SCHEMA.SOMETABLE, TESTCOLB = value abc
operation: UPDATE, on table "SCHEMA.SOMETABLE":
SCHEMA.SOMETABLE, TESTCOLA = value 456
SCHEMA.SOMETABLE, TESTCOLB = value def
operation: UPDATE, on table "SCHEMA.SOMETABLE":
SCHEMA.SOMETABLE, TESTCOLA = value 789
SCHEMA.SOMETABLE, TESTCOLB = value ghi
```

Coding a custom handler in Java

A custom handler can be implemented by extending AbstractHandler:

```
import com.goldengate.atg.datasource.*;
import static com.goldengate.atg.datasource.GGDataSource.Status;

public class SampleHandler extends AbstractHandler {

    @Override
    public void init(DsConfiguration conf, DsMetaData metaData) {
        super.init(conf, metaData);
        // ... do additional config...
    }

    @Override
    public Status operationAdded(DsEvent e, DsTransaction tx, DsOperation
    op) { ... }

    @Override
    public Status transactionCommit(DsEvent e, DsTransaction tx) { ... }

    @Override
    public Status metaDataChanged(DsEvent e, DsMetaData meta) { .... }

    @Override
    public void destroy() { /* ... do cleanup ... */ }

    @Override
    public String reportStatus() { return "status report..."; }
}
```

When a transaction is processed from the Extract, the order of calls into the handler is as follows:

1. Initialization:
    - First, the handler is constructed;
    - Next, all the "setters" are called on the instance with values from the property file;
    - Finally, the handler is initialized; the init(...) method is called before any transactions are received. It is important that the init(...) method call super.init(....) to properly initialize the base class.

2. Metadata is received: if the user exit is processing an operation on a table not yet seen during this run, a metadata event is fired, and the metadataChanged(...) method is called. Typically, there is no need to implement this method. The DsMetaData is automatically updated with new data source metadata as it is received.
3. A transaction is started: a transaction event is fired, causing the transactionBegin(...) method on the handler to be invoked (not shown). This is typically not used, since the transaction has zero operations at this point.
4. Operations are added to the transaction, one after another; this causes the operationAdded(...) method to be called on the handler for each operation added. The containing transaction is also passed into the method, along with the data source metadata (containing all table metadata seen thus far). Note that the transaction has not yet been committed, and could be aborted before the commit is received.

   Each operation contains the column values from the transaction (possibly just the changed values, if Extract is processing with compressed updates. The column values may contain both before and after values.
5. The transaction is committed; this causes the transactionCommit(...) method to be called.
6. Periodically, reportStatus may be called; it is also called at process shutdown. Typically, this displays the statistics from processing (number of operations/transactions processed, etc).

Below is a complete example of a simple printer handler, which just prints out very basic event information for transactions, operations and metadata. Note that the handler also has a property `myoutput` for setting the output filename; this can be set in the Java application properties file as follows:

```
gg.handlerlist=sample
# set properties on 'sample'
gg.handler.sample.type=sample.SampleHandler
gg.handler.sample.myoutput=out.txt
```

To use the custom handler,

1. Compile the class
2. Include the class in the application classpath,
3. Add the handler to the list of active handlers in the Java application properties file.

To compile the handler, include the Oracle GoldenGate for Java jars in the classpath and place the compiled .class files in {gg_install_dir}/javaue/resources/classes:

```
javac -d {gg_install_dir}/dirprm
-classpath ggjava/ggjava.jar SampleHandler.java
```

The resulting class files would be located in resources/classes, in correct package structure, such as:

```
{gg_install_dir}/dirprm/sample/SampleHandler.class
```

NOTE For any Java application development beyond "hello world" examples, either Ant or Maven would be used to compile, test and package the application. The examples showing javac are for illustration purposes only.

Alternatively, custom classes can be put into a jar and included in the classpath. Either include the custom jar file(s) in the JVM classpath via the user exit properties (using java.class.path in the javawriter.bootoptions property), or by setting the Java application properties file to include your custom jar:

```
# set properties on 'one'
gg.handler.one.type=sample.SampleHandler
gg.handler.one.myoutput=out.txt
gg.classpath=/path/to/my.jar,/path/to/directory/of/jars/*
```

The classpath property can be set on any handler to include additional individual jars, a directory (which would contain resources or unjarred class files) or a whole directory of jars. To include a whole directory of jars, use the Java 6 style syntax:

```
c:/path/to/directory/* (or on Unix: /path/to/directory/* )
```

Only the wildcard * can be specified; a file pattern cannot be used. This automatically matches all files in the directory ending with the .jar suffix. To include multiple jars or multiple directories, you can use the system-specific path separator (on Unix, the colon and on Windows the semicolon) or you can use platform-independent commas, as shown above.

If the handler requires many properties to be set, just include the property in the parameter file, and your handler's corresponding "setter" will be called. For example:

```
gg.handler.one.type=com.mycompany.MyHandler
gg.handler.one.myOutput=out.txt
gg.handler.one.myCustomProperty=12345
```

The above example would invoke the following methods in the custom handler:

```
public void setMyOutput(String s) {
    // use the string...
} public void setMyCustomProperty(int j) {
    // use the int...
}
```

Any standard Java type may be used, such as int, long, String, boolean, etc. For custom types, you may create a custom property editor to convert the String to your custom type.

Additional resources

There is Javadoc available for the Java API. The Javadoc has been intentionally reduced to a set of core packages, classes and interfaces in order to only distribute the relevant interfaces and classes useful for customization and extension.

In each package, some classes have been intentionally omitted for clarity. The important classes are:

- com.goldengate.atg.datasource.DsTransaction: represents a database transaction. A transaction contains zero or more operations.
- com.goldengate.atg.datasource.DsOperation: represents a database operation (insert, update, delete). An operation contains zero or more column values representing the data-change event. Columns indexes are offset by zero in the Java API.

- com.goldengate.atg.datasource.DsColumn: represents a column value. A column value is a composite of a before and an after value. A column value may be 'present' (having a value or be null) or 'missing' (is not included in the source trail).
  - com.goldengate.atg.datasource.DsColumnComposite is the composite
  - com.goldengate.atg.datasource.DsColumnBeforeValue is the column value before the operation (this is optional, and may not be included in the operation)
  - com.goldengate.atg.datasource.DsColumnAfterValue is the value after the operation
- com.goldengate.atg.datasource.meta.DsMetaData: represents all database metadata seen; initially, the object is empty. DsMetaData contains a hash map of zero or more instances of TableMetaData, using the TableName as a key.
- com.goldengate.atg.datasource.meta.TableMetaData: represents all metadata for a single table; contains zero or more ColumnMetaData.
- com.goldengate.atg.datasource.meta.ColumnMetaData: contains column names and data types, as defined in the database and/or in the Oracle GoldenGate source definitions file.

See the Javadoc for additional details.

CHAPTER 11
Troubleshooting

Perform the checks listed in error handling. If you do not succeed, contact Oracle GoldenGate Support.

Error handling

There are three types of errors that could occur in the operation of Oracle GoldenGate for Java:

- The Extract process running the user exit or VAM does not start or abends
- The process runs successfully, but the data is incorrect or nonexistent

If the Extract process does not start or abends, check the error messages in order from the beginning of processing through to the end:

1. Check the Oracle GoldenGate event log for errors, and view the Extract report file:

```
GGSCI> VIEW GGSEVT
GGSCI> VIEW REPORT {extract name}
```

2. Check the applicable log file.

   For the user exit:

   - Look at the last messages reported in the log file for the user exit library. The file name is the log file prefix (`log.logname`) set in the property file and the current date.

     ```
     shell> more {log.logname}_{yyyymmdd}.log
     ```

   Note: This is only the log file for the shared library, not the Java application.

3. If the user exit or VAM was able to launch the Java runtime, then a log4j log file will exist.

   The name of the log file is defined in your log4j.properties file. By default, the log file name is ggjava-<version>-log4j.log, where <version> is the version number of the jar file being used. For example:

   ```
   shell> more ggjava-*log4j.log
   ```

   To set a more detailed level of logging for the Java application, either:

   - Edit the current log4j properties file to log at a more verbose level or

- Re-use one of the existing log4j configurations by editing properties file:

```
{javawriter or jvm}.bootoptions=-Djava.class.path=ggjava/ggjava.jar
-Dlog4j.configuration=debug-log4j.properties -Xmx512m
```

These pre-configured log4j property files are found in the classpath, and are installed in:

```
./ggjava/resources/classes/*log4j.properties
```

4. If one of these log files does not reveal the source of the problem, run the Extract process directly from the shell (outside of GGSCI) so that `stderr` and `stdout` can more easily be monitored and environmental variables can be verified. For example:

```
shell> EXTRACT PARAMFILE dirprm/javaue.prm
```

If the process runs successfully, but the data is incorrect or nonexistent, check for errors in any custom filter, formatter or handler you have written for the user exit.

To restart the user exit Extract from the beginning of a trail, see page 67.

For further information on troubleshooting the core Oracle GoldenGate software, see the *Oracle GoldenGate Troubleshooting and Performance Tuning Guide*.

Reporting issues

If you have a support account for Oracle GoldenGate, please submit a support ticket. Please include:

- Operating system and Java versions

  The version of the Java Runtime Environment can be displayed by:

  ```
  $ java -version
  ```

- Configuration files:
  - Parameter file for the Extract running the user exit
  - All properties files used, including any JMS or JNDI properties files
  - Velocity templates for the user exit

- Log files:

  In the Oracle GoldenGate install directory, all .log files: the Java log4j log files and the user exit or VAM log file.

Index

A

abend 92
ActiveMQ 70
application
restart 67
start 18, 67

B

boot options
JVM 39, 75

C

checkpoint
user exit 67
column data 22, 23, 24
fixed width parsing 24
comma-separated values 79
comments
entering 37, 73
to identify key columns 27
to specify date format 26
configuration options 13
configure
data pump 64
event handlers 69
Java handlers 66
JRE 64
VAM Extract 19
connection factory 84
copybook
definition for fixed width parsing 25
for source and target definitions 22
CSV format 79
CUSEREXIT 65
custom formatters 69
custom Java code 13

D

data definitions
how to specify 22
data pump
configure 64
Defgen 18
delimited message
basis for parsing 27
format 27
metadata columns 28
parsing properties 46
parsing rules 28
dirdat directory 18
Djava.class.path 75
dll 13
dynamically linked library 13

E

errors 92
ETL tools 11
event filters 69
event handlers 69
configuring 69
EXCLUDEUSER parameter 25

Extract
- adding the VAM Extract 19
- configuring for the VAM 19
- parameters for the user exit 65
- parameters for the VAM 19
- running user exit 18
- starting the Java application 18
- the VAM Extract 18
- user exit Extract 64

F

file handler 71

file writer
- properties 81

filewriter handler 69

fixed width message
- basis for parsing 25
- defining the header 25
- key identification 27
- parsing properties 42
- table name 26
- timestamp formats 26

fixed-format 79

flat file integration 11

formatters
- custom 69

formatting 72, 78

G

Gendef utility 22, 36

general properties
- Java framework 84
- user exit 75

GETENV function 25

ggjava_ue.dll 17

ggjava_vam.dll 17

ggjava.jar 18

ggue.jar 18

GoldenGate
- installation directory 18

H

handler
- event 69
- file 71
- filewriter 69
- JMS 70
- properties 77

handlers
- configuring 69

header
- defining for fixed width message 25

I

installation directory structure
- GoldenGate 18

installing 15, 19
- GoldenGate 16
- Java 15
- Java application 18
- User Exit for Java 15, 64

issues
- reporting 93

J

jars 12, 18

Java API 11, 12

Java application
- installing 18
- properties 77

Java code 87

Java Development Kit 15

Java handlers
- configure 66

Java jars 18

Java libraries 12

Java Message Service (JMS) 12

Java Native Interface (JNI) 12

Java Runtime Environment (JRE) 15

Java SDK 15

Java User Exit
checkpoint 67
installing 15, 19
running 67
java version 16
Java virtual machine (JVM) 15
JAVA_HOME 15, 16
Java, installing 15
java.class.path 18, 64
javaue.propertie 18
javawriter.bootoptions 18
JBoss 70
JDK 15
JMS 12
connecting to 20
properties 38, 70
standard settings 83
jms 71
JMS handler 14, 70
properties 82
JMS messages
retrieving 21
JMS provider 70
JMS queue or topic 83
jms_map 71
jms_text 71
JNDI 20, 40
properties 41, 71
JNI 12
JRE 15
configure 64
JVM 15
JVM boot options 75
jvm.dll 15

K

key identification
for fixth width messages 27

L

LD_LIBRARY_PATH 16
libggjava_ue.so 17
libggjava_vam.so 17
Linux 16
log4j.configuration 64
logging properties 37, 73

M

Manager parameter file 16
MapMessage 69
Message Capture
running 22
message format 13
metadata columns
delimited message 28

O

operation type 22, 23, 30
for XML parsing 30
mapping 26
optype
specifying for fixed width parsing 26
Oracle GoldenGate
documentation 14
Oracle GoldenGate for Java 12
output 72, 78

P

parameters
VAM Extract 19
PARAMS option 20
parser
required data 23
role of 22
types 22
PATH environmental variable 15
PIC
translations for fixth width messages 27

properties 37, 73
- delimited message parsing 46
- file writer 81
- fixed width message parsing 42
- handlers 77
- Java application 77
- Java framework 84
- JMS 70
- JMS handler 82
- JNDI 41, 71
- logging 37, 73
- User Exit 37, 73
- XML message parsing 54

property file 13

Q

queue 83

R

reporting 72, 76
- issues 93

restart
- application 67

running
- Java User Exit 67

S

sequence identifier 22, 23

SETENV 65

Solace 70

source definitions file 19, 42, 46, 54
- generating 22, 36

sourcedefs
- type of fixed schema 42

SOURCEDEFS parameter 65

standard JMS settings 83

start
- application 18, 67

statistics 76

subdirectories
- creating 16

T

TABLE 65

table name 22, 23, 28, 29, 33
- defining for fixed width message 26
- for delimited parsing 28
- for XML parsing 30

TextMessage 69

TIBCO 70

timestamp 22, 23, 25, 26, 28, 30, 32, 33
- formats for fixed width message 26

topic 83

trail 64

TRANLOGOPTIONS
- GETMETADATAFROMVAM option 20
- VAMCOMPATIBILITY option 20

transaction
- specifying boundary for 24, 33

transaction identifier 22, 23
- for XML parsing 33

transaction indicator 22, 24, 28, 33

transaction name 22, 25

transaction owner 22, 25

troubleshooting 92

U

UNIX 16

unzip 17

User Exit
- properties 37, 73
- running 67

User Exit for Java
- checkpoint 67

V

VAM parameter 19

Velocity template 13, 78, 87

W

WebLogic 70

Windows 15

X

XML 13

XML message

basis for parsing 29

column rules 34

formatted in dynamic XML 29

formatted in static XML 29

operation rules 33

parsing properties 54

parsing rules 30

supported XPath expressions 31

transaction rules 33