Oracle® GoldenGate for Java

Administration Guide

Version 3.0

October 2009

Administration Guide, version 3.0

Contents

CHAPTER 1 Introduction

This guide covers:

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

Oracle GoldenGate Transactional Data Management

The core Oracle GoldenGate product is a Transactional Data Management (TDM) platform that:

- Captures transactional changes from a source database by reading the database transaction log
- Sends and queues these changes on local or remote disk, as a set of database-independent binary 'trail' files
- Optionally transforms the source data
- Applies the transactions in the trail to a target system: a database, JMS provider or other messaging system or custom application.

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

Integration Options

The transactional changes may be applied to targets other than a relational database: for example, ETL tools (DataStage, Ab Initio, Informatica), messaging systems (JMS), 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; for example: delimiter separated values, length delimited values, binary, etc. Near real-time feeds to these systems are accomplished by decreasing the time window for batch file rollover to minutes or even seconds.
- *Messaging systems*: 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 for Java

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** 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).

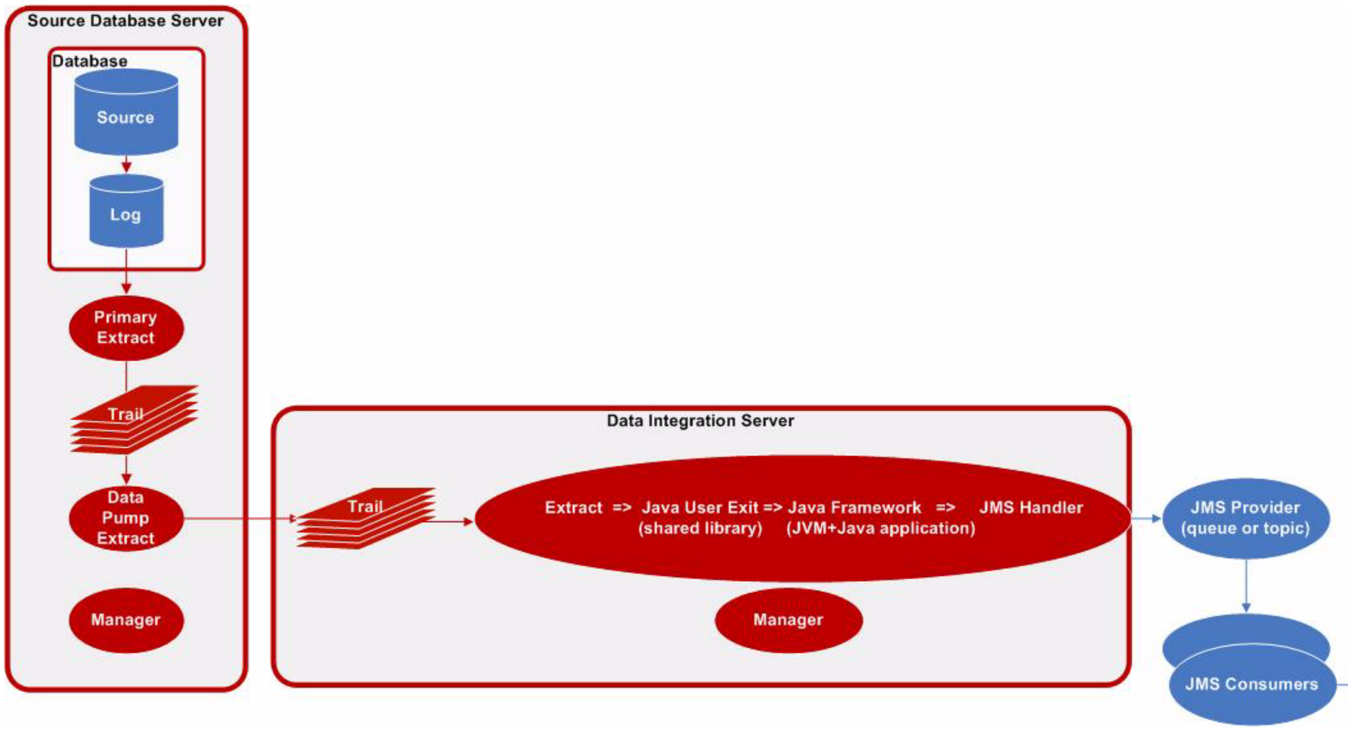

Figure 1 Configuration using the JMS Handler

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, see the Oracle GoldenGate documentation:

- *Oracle GoldenGate Installation and Setup Guides*: One for each database supported by GoldenGate.
- *Oracle GoldenGate Administration Guide*: Introduces Oracle GoldenGate components and explains how to plan for, configure, and implement Oracle GoldenGate.
- *Oracle GoldenGate Reference Guide*: Provides detailed information about Oracle GoldenGate parameters, commands, and functions.
- *Oracle GoldenGate Troubleshooting and Performance Tuning Guide*: Provides suggestions for improving the performance of Oracle GoldenGate in different situations, and provides solutions to common problems.

These manuals are available for download from http://support.goldengate.com (support login ID and password required).

The Oracle GoldenGate website (http://www.goldengate.com) has white papers, web seminars and user conference information. The support site's Knowledge Base is an online resource containing the latest information on bug fixes and configuration notes.

CHAPTER 2 Installing Java and Oracle GoldenGate Software

Before running Oracle GoldenGate for Java, you must install

1. Java (JDK or JRE) version 1.5 or later
2. Oracle GoldenGate version 10.0.0.4 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). However, the dynamically linked library is specific to the version of Java being used.

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 process(es). 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 has been installed, configure the PATH to find the (1) JRE and (2) the 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 32bit or 64bit JVM is being used.

Installing Oracle GoldenGate 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.

- Create the required subdirectories by starting GGSCI and running:
  ```
  GGSCI> CREATE SUBDIRS
  ```
- 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
  ```

The following outlines the standard steps required to configure Oracle GoldenGate to capture changes from a source database, write these deltas to a trail file on disk, and use a data pump to read in this trail file. This is just one possible configuration; for details and other configuration options, see the Oracle GoldenGate documentation. In these PDF guides you will find complete instructions and explanations, including important operating system and database-specific considerations. Once Oracle GoldenGate is installed and

Java is available, then Oracle GoldenGate for Java can be implemented. Oracle GoldenGate for Java runs as a user exit in the data pump process, reading a local trail file.

Prepare the database

The database must be configured to write the necessary data to the transaction log for Extract to capture. This is a database-specific process; see the *Oracle GoldenGate Installation and Setup Guide* for each database. This typically only has to be done once and only once for the entire database instance.

As an example, for Oracle, the system must be altered to enable supplemental logging:

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
SQL> ALTER SYSTEM SWITCH LOGFILE;
```

Prepare the tables

For every table that is to be captured by Oracle GoldenGate, at least the primary keys or a unique index must be logged to the transaction log. Optionally, other data may be also logged, such as full updates (not just the changed columns).

To force the primary keys to be logged for all tables in a particular schema (for example, a schema called GGS), run the following in GGSCI (for systems other than Oracle, provide an ODBC data source name):

```
GGSCI> DBLOGIN USERID myuser, PASSWORD mypassword
GGSCI> ADD TRANDATA GGS.*
GGSCI> INFO TRANDATA GGS.*
```

If the tables are dropped and re-added, "trandata" will have to be enabled once again. For syntax details and options for "trandata" related commands, use the GGSCI built-in help command:

```
GGSCI> HELP ADD TRANDATA
```

Primary capture on the database server

Use GGSCI to set up a primary Extract on the database server. A primary Extract captures transactions from the database transaction log for the specified tables (as defined in the parameter file for the Extract process). To add the Extract via GGSCI:

```
GGSCI> ADD EXTRACT capture, TRANLOG, BEGIN NOW
GGSCI> ADD EXTTRAIL ./dirdat/aa, EXTRACT capture, MEGABYTES 20
```

The parameter file for the Extract would look something like the following for Oracle:

```
EXTRACT capture
USERID mydblogin, PASSWORD mydbpass
-- SOURCEDEFS mydefs.def
EXTTRAIL ./dirdat/aa
EOFDELAY 2
-- GETUPDATEBEFORES
-- NOCOMPRESSUPDATES
TABLE GGS.*;
```

NOTE For databases other than Oracle, the login information would include an ODBC data source name, e.g. SOURCEDB mydsn, USERID myuser, PASSWORD mypass)

If there changes in the database that are not captured by this primary Extract, then these changes will not be available to the Java application further down the line. For example, by default, the before values (the values before the update happens) are *not* captured, and only compressed updates and compressed deletes are captured (only the changed columns for updates, and only the primary key columns for delete operations). In the example parameter file above, un-comment the two parameters to include before images and uncompressed updates.

NOTE To capture uncompressed updates (NOCOMPRESSUPDATES), the database will have to be configured to log these additional columns to the transaction log. An alternative to forced logging is to fetch (query) the values from the database. See the *Oracle GoldenGate Reference Guide* for details.

Secondary data pump on database server (optional)

In addition to the primary Extract running on the database host, you may optionally set up a secondary data pump to send the data to a remote system.

```
GGSCI > ADD EXTRACT mypump, EXTTRAILSOURCE ./dirdat/aa
GGSCI > ADD RMTTRAIL ./dirdat/bb, EXTRACT mypump, MEGABYTES 20
```

The parameter file for a data pump Extract is usually quite trivial, unless additional processing is included such as filtering, transformations, or running user exit logic. A simple data pump is often configured for PASSTHRU mode to disable filtering, transformations and user exit logic:

```
EXTRACT mypump
USERID mydblogin, PASSWORD mydbpass
-- SOURCEDEFS mydefs.def
RMTHOST mytargethost, MGRPORT 7802
RMTTRAIL ./dirdat/bb
PASSTHRU
TABLE GGS.*;
```

A data pump on the database server is an optional, but typical (and suggested) configuration. If a data pump is not used, then the primary Extract would not use a local trail (EXTTRAIL), but rather send the changes to a remote trail on a remote host (RMTHOST / RMTTRAIL).

Although a user exit always runs in a data pump, the user exit may run either on the database server or on a remote host. The following are factors to consider when choosing whether to run a user exit in a data pump on the database host or a separate host:

- What is the load added to the system by the user exit?

  If the load is significant, you may want to move this processing to an intermediate host, to not impact the performance of the database.

- What is the reliability of the network between the source database server and the target systems?

If the network is not reliable, you may want to have the primary Extract write to a local trail, and use a data pump to process this trail. The primary Extract should not fall behind in processing due to extended network outages.

- Can additional software be installed on the database host?

  If Java is not installed or an older version can not easily be updated, then Oracle GoldenGate for Java may have to run on a separate host.

See the *Oracle GoldenGate Administration Guide* for additional considerations in choosing your architecture.

CHAPTER 3

Installing Oracle GoldenGate for Java

Oracle GoldenGate for Java includes two components:

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

Installing the Shared (Dynamically Linked) Library

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

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

Unzip into the Oracle GoldenGate directory

Oracle GoldenGate for Java is distributed as a zip file containing the platform-dependent user exit library (JavaUserExit.so or JavaUserExit.dll) and a associated properties file. The Java jars (platform independent) are also distributed as a zip file.

NOTE The platform-independent Java jars and the platform-dependent user exit 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.

Simply extract these zip files into Oracle GoldenGate's installation directory, as shown below. The installation location of the User Exit shared library and the related Java jar files is configurable; the following example configuration 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`
`| . . .`
	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.

`|-[dirdat]`
`| |aa000000`
`| |aa000001`
`| . . .`
`|-[dirdef]`
`| |-mysrcdefs.def`
`|-[dirprm]`
`| |-javaue.prm`
`| . . .`
	The Extract running the user exit is configured as a data pump, consuming trail data (produced by the primary Extract) in the dirdat directory.
	The metadata (column names, data types) for the trail data can come from a sourcedefs file (e.g. mysrcdefs.def, generated by DEFGEN) or from the database.
	The Extract parameter (javaue.prm) file specifies the User Exit library to load.

`|-JavaUserExit.dll`
`|-cuserexit.properties`
`| . . .`
	The user exit shared library: JavaUserExit.dll (Windows) or JavaUserExit.so (UNIX/Linux). cuserexit.properties is a sample user exit properties file.

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

Configure the JRE (in the user exit propertiesfile)

Modify the user exit properties file to point to the location of the Oracle GoldenGate for Java main jar (ggue.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=javaue/ggue.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 (ggue.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 preconfigured 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 25 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.

```
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 sourcedefs file. This metadata defines the column names and data types in the trail being read (./dirdat/aa).
`SETENV (GGS_USEREXIT_CONF = "dirprm/cuserexit.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.

Table 2 User Exit Extract Parameters

Parameter	Explanation
`SETENV (GGS_JAVAUSEREXIT_CONF = "/dirprm/javaue.properties")`	(Optional) The Java properties file, as a classpath resource. The classpath is defined in the C User Exit properties file. This example places the properties file in the dirpm directory, assuming the Oracle GoldenGate installation directory is in the classpath.
`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` ◆ The callback function name - must be uppercase CUSEREXIT. ◆ 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/cuserexit.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 given is a path to a file found in the classpath.

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=javaue/ggue.jar
```

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 4
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/tcustord trail is used (matching the demo SQL provided with the Oracle GoldenGate software download), along with a sourcedef 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 sourcedef 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 43:

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 5 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=/dirprm/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=/dirprm/foo.properties`.

JMS Handler

The main Java property file `ggue.properties` 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 (javaue) contains the core application jars (ggue.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	Can be set to queue or topic.
gg.jmshandler.destination=queue/A	The destination name for the queue or topic is configurable 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

Using the file handler is quite simple; it 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 38.

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 35.

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 6 Properties

The following section defines the options available for configuration of the two property files for Oracle GoldenGate for Java:

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

Both should be placed in <Oracle GoldenGate installation directory>/dirprm. and environmental variables set:

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

User Exit Properties

All properties in the property file are of the form: `fully.qualified.name=value`

The value may be a single string, integer, or boolean, or could be 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 (useful for testing various configurations). 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 properties

Logging is standard to many Oracle GoldenGate user exits solutions and is controlled by the following properties.

`goldengate.log.logname`
Takes any valid string as the prefix to the log file name. The log file produced has the current date appended to it, formatted as `yyyymmdd`, suffixed by the extension `.log`. For example, the following produces a filename such as: `javawriter_20071230.log`

```
# log file prefix
goldengate.log.logname=javawriter
```

The log file will roll over each day, independent of the process restarting.

`goldengate.log.level`
Set the overall log level for all modules. 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, but would not impeded performance. If the level is switch 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
goldengate.log.level=INFO
```

`goldengate.log.{tostdout,tofile}`
Boolean properties that determine if logged messages are written to stdout, and/or to a specified log file. Most Oracle GoldenGate processes run as background processes, so stdout is generally not a useful option. For example:

```
# output params, to stdout and/or to file
goldengate.log.tostdout=false
goldengate.log.tofile=true
```

`goldengate.log.modules, goldengate.log.level.{module}`
This is typically for advanced debugging only; the log level of individual source modules that comprise the user exit can be specified individually. 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 prompted to do so by a support engineer.

Generic properties

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

`goldengate.userexit.writers`
String value specifying name of 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`
String value for the prefix to be 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 determine now the Java Runtime Environment will be configured. 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`
Specify 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, ggue.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=./javaue/ggue.jar
-Dlog4j.configuration=my-log4j.properties -Xmx512m
```

The log4j.configuration property could be a fully qualified URI 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`
Boolean (true/false) value which, if true, will output statistics to the Oracle GoldenGate report file and to the user exit log files. For example:

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

`javawriter.stats.full`
Boolean (true/false) value which, if true, will output Java side statistics in addition to the C side statistics. Java side statistics are more detailed but also involve some additional overhead. Therefore if stats are reported quite often and a less detailed summary suffices, it is recommended to set the stats.full property to false. For example:

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

`javawriter.stats.{time, numrecs}`
The default is to report statistics every hour or every 10000 records (which ever occurs first). A time interval can be specified in seconds, after which statistics will be reported. Also, the number of records may also be specified, after which the statistics will be reported. 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}
```

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 is customizable)
- **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 `{gg_install_dir}/dirprm`.

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* `tabledt_modifydate` in all schemas:

```
gg.handler.myhandler.excludeTables=dt_modifydate
```

`gg.handler.<name>.mode`
`gg.handler .<name>.format.mode`
Specify 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-formatted 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`
Delimiter to use between fields (set to no value in order to have no delimiter used). For example: `delim=,`

`quote`
If column values should be quoted, identify the quote character here. For example: quote='

`metacols`
These metacolumn values 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`
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`
Header meta chars (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`
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 (respectively):

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

`endOfLine`
Set end-of-line character to:

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

For example: `endOfLine=CR`

`justify`
Set fixed fields to left or right-justify. For example: `justify=left`

`includeBefores`
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`
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`

Set to true or false to determine if the file should be appended to or overwritten upon restart.

`gg.handler.{name}.rolloverSize`
If using the file handler, this is the size of the file before a rollover should be attempted. The filesize 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 10Mb.

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 Oracle GoldenGate install 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 only brief descriptions (for further details, see "Standard JMS settings" on page 33):

destination: queue or topic name

queueOrTopic: [queue | topic]

user : username 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 is 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 – additional directories or jars to add to classpath.

gg.report.format – template to use for customizing the report format.

CHAPTER 7

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 GoldenGatefor Java jars in the classpath and place the compiled .class files in {gg_install_dir}/dirprm:

```
javac -d {gg_install_dir}/dirprm
-classpath javaue/ggue.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 prototyping 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'
#foreach( $op in $tx )
operation: $op.sqlType, on table "$op.tableName":
#foreach( $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 javaue/ggue.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 developement 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 so as 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 sourcedefs file.

See the Javadoc for additional details.

CHAPTER 8 Troubleshooting

Perform the checks listed in error handling. If you do not succeed, contact Oracle 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 does not start or abends
- The process runs successfully, but the data is incorrect or nonexistent

If the user exit 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 JAVAUE
   ```

2. Look at the last messages reported in the log file for the user exit library; the file name is the <log file prefix> defined in javawriter.properties and the current date:

   ```
   shell> more cuserexit_<yyyymmdd>.log
   ```

   Note: This is only the log file for the JNI (native) shared library, not the Java application.

3. If the user exit was able to launch the Java runtime, then a log4j log file will be generated from Java.

   The name of the log file is defined in your log4j.properties file pointed to by the user exit properties. By default, the log file name is ggue-<version>-log4j.log, where <version> is the version number of the jar file being used. For example:

   ```
   shell> more ggue-2.2.6-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 javawriter.properties:

     ```
     javawriter.bootoptions=-Djava.class.path=./javaue/ggue.jar
     -Dlog4j.configuration=debug-log4j.properties –Xmx512m
     ```

   These pre-configured log4j property files are found in the classpath, and are installed in:

   ```
   ./javaue/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:

   ```
   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.

To restart the user exit Extract from the beginning of a trail, see page 19.

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 by from the support site (http://support.goldengate.com). Please include:

- Oracle GoldenGate and JavaUser Exit versions
- Operating system and Java versions

  The version of the Oracle GoldenGate for Java jars can be displayed by:

  ```
  $ java -jar ggue.jar
  ```

  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
- Log files:

  In the Oracle GoldenGate install directory, all .log files: the Java log4j log files and the user exit log file.

Index