Oracle® Fusion Middleware

Administering Oracle GoldenGate for Big Data

Release 12*c* (12.3.1.1)
E89479-01
August 2017

Oracle Fusion Middleware Administering Oracle GoldenGate for Big Data, Release 12*c* (12.3.1.1)

E89479-01

Copyright © 2015, 2017, 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 is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware 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 that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about 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 unless otherwise set forth in an applicable agreement between you and Oracle. 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, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience xiii
Documentation Accessibility xiii
Related Information xiii
Conventions xiv

Part I Understanding Oracle GoldenGate for Big Data

1 Understanding the Java Adapter and Oracle GoldenGate for Big Data

1.1 Overview 1-1
1.1.1 Oracle GoldenGate Features 1-1
1.1.2 Adapter Integration Options 1-1
1.1.2.1 Capturing Transactions to a Trail 1-2
1.1.2.2 Applying Transactions from a Trail 1-2
1.2 Using Oracle GoldenGate Java Adapter Properties 1-3
1.2.1 Values in Property Files 1-3
1.2.2 Location of Property Files 1-4
1.2.3 Using Comments in the Property File 1-4
1.2.4 Variables in Property Names 1-4
1.3 Oracle GoldenGate Documentation 1-5

2 Introducing the Java Adapter

2.1 Oracle GoldenGate VAM Message Capture 2-1
2.1.1 Message Capture Configuration Options 2-1
2.1.2 Typical Configuration 2-1
2.2 Oracle GoldenGate Java Delivery 2-2
2.3 Delivery Configuration Options 2-3

3 Configuring Logging

3.1 Oracle GoldenGate Java Adapter Default Logging 3-1
3.1.1 Default Implementation Type 3-1
3.1.2 Default Message Logging 3-1
3.1.3 Log File Name 3-1
3.1.4 Logging Problems 3-1
3.2 Recommended Logging Settings 3-1
3.2.1 Changing to the Recommended Logging Type 3-2

Part II Capturing JMS Messages

4 Configuring Message Capture

4.1 Configuring the VAM Extract 4-1
4.1.1 Adding the Extract 4-1
4.1.2 Configuring the Extract Parameters 4-2
4.1.3 Configuring Message Capture 4-2
4.2 Connecting and Retrieving the Messages 4-2
4.2.1 Connecting to JMS 4-2
4.2.2 Retrieving Messages 4-3
4.2.3 Completing the Transaction 4-3

5 Parsing the Message

5.1 Parsing Overview 5-1
5.1.1 Parser Types 5-1
5.1.2 Source and Target Data Definitions 5-2
5.1.3 Required Data 5-2
5.1.3.1 Transaction Identifier 5-2
5.1.3.2 Sequence Identifier 5-2
5.1.3.3 Timestamp 5-3
5.1.3.4 Table Name 5-3
5.1.3.5 Operation Type 5-3
5.1.3.6 Column Data 5-4
5.1.4 Optional Data 5-4
5.1.4.1 Transaction Indicator 5-4
5.1.4.2 Transaction Name 5-4
5.1.4.3 Transaction Owner 5-4
5.2 Fixed Width Parsing 5-5
5.2.1 Header 5-5

5.2.1.1 Specifying Compound Table Names 5-6
5.2.1.2 Specifying timestamp Formats 5-6
5.2.1.3 Specifying the Function 5-6
5.2.2 Header and Record Data Type Translation 5-7
5.2.3 Key identification 5-7
5.2.4 Using a Source Definition File 5-7
5.3 Delimited Parsing 5-9
5.3.1 Metadata Columns 5-9
5.3.2 Parsing Properties 5-10
5.3.2.1 Properties to Describe Delimiters 5-10
5.3.2.2 Properties to Describe Values 5-10
5.3.2.3 Properties to Describe Date and Time 5-11
5.3.3 Parsing Steps 5-11
5.4 XML Parsing 5-11
5.4.1 Styles of XML 5-11
5.4.2 XML Parsing Rules 5-12
5.4.3 XPath Expressions 5-13
5.4.3.1 Supported Constructs: 5-13
5.4.3.2 Supported Expressions 5-14
5.4.3.3 Obtaining Data Values 5-14
5.4.4 Other Value Expressions 5-15
5.4.5 Transaction Rules 5-15
5.4.6 Operation Rules 5-16
5.4.7 Column Rules 5-17
5.4.8 Overall Rules Example 5-18
5.5 Source Definitions Generation Utility 5-19

6 Message Capture Properties

6.1 Logging and Connection Properties 6-1
6.1.1 Logging Properties 6-1
6.1.1.1 gg.log 6-1
6.1.1.2 gg.log.level 6-1
6.1.1.3 gg.log.file 6-2
6.1.1.4 gg.log.classpath 6-2
6.1.2 JMS Connection Properties 6-2
6.1.2.1 jvm.boot options 6-2
6.1.2.2 jms.report.output 6-3
6.1.2.3 jms.report.time 6-3
6.1.2.4 jms.report.records 6-3
6.1.2.5 jms.id 6-4

6.1.2.6 jms.destination 6-4
6.1.2.7 jms.connectionFactory 6-4
6.1.2.8 jms.user, jms.password 6-4
6.1.3 JNDI Properties 6-5
6.2 Parser Properties 6-5
6.2.1 Setting the Type of Parser 6-5
6.2.1.1 parser.type 6-5
6.2.2 Fixed Parser Properties 6-6
6.2.2.1 fixed.schematype 6-6
6.2.2.2 fixed.sourcedefs 6-7
6.2.2.3 fixed.copybook 6-7
6.2.2.4 fixed.header 6-7
6.2.2.5 fixed.seqid 6-7
6.2.2.6 fixed.timestamp 6-8
6.2.2.7 fixed.timestamp.format 6-8
6.2.2.8 fixed.txid 6-8
6.2.2.9 fixed.txowner 6-9
6.2.2.10 fixed.txname 6-9
6.2.2.11 fixed.optype 6-9
6.2.2.12 fixed.optype.insertval 6-9
6.2.2.13 fixed.optype.updateval 6-9
6.2.2.14 fixed.optype.deleteval 6-10
6.2.2.15 fixed.table 6-10
6.2.2.16 fixed.schema 6-10
6.2.2.17 fixed.txind 6-10
6.2.2.18 fixed.txind.beginval 6-10
6.2.2.19 fixed.txind.middleval 6-11
6.2.2.20 fixed.txind.endval 6-11
6.2.2.21 fixed.txind.wholeval 6-11
6.2.3 Delimited Parser Properties 6-11
6.2.3.1 delim.sourcedefs 6-12
6.2.3.2 delim.header 6-12
6.2.3.3 delim.seqid 6-13
6.2.3.4 delim.timestamp 6-13
6.2.3.5 delim.timestamp.format 6-13
6.2.3.6 delim.txid 6-14
6.2.3.7 delim.txowner 6-14
6.2.3.8 delim.txname 6-14
6.2.3.9 delim.optype 6-14
6.2.3.10 delim.optype.insertval 6-14
6.2.3.11 delim.optype.updateval 6-15

6.2.3.12 delim.optype.deleteval 6-15
6.2.3.13 delim.schemaandtable 6-15
6.2.3.14 delim.schema 6-15
6.2.3.15 delim.table 6-15
6.2.3.16 delim.txind 6-16
6.2.3.17 delim.txind.beginval 6-16
6.2.3.18 delim.txind.middleval 6-16
6.2.3.19 delim.txind.endval 6-16
6.2.3.20 delim.txind.wholeval 6-16
6.2.3.21 delim.fielddelim 6-17
6.2.3.22 delim.linedelim 6-17
6.2.3.23 delim.quote 6-17
6.2.3.24 delim.nullindicator 6-17
6.2.3.25 delim.fielddelim.escaped 6-17
6.2.3.26 delim.linedelim.escaped 6-18
6.2.3.27 delim.quote.escaped 6-18
6.2.3.28 delim.nullindicator.escaped 6-19
6.2.3.29 delim.hasbefores 6-19
6.2.3.30 delim.hasnames 6-20
6.2.3.31 delim.afterfirst 6-20
6.2.3.32 delim.isgrouped 6-20
6.2.3.33 delim.dateformat | delim.dateformat.table | delim.dateform.table.column 6-20
6.2.4 XML Parser Properties 6-21
6.2.4.1 xml.sourcedefs 6-22
6.2.4.2 xml.rules 6-22
6.2.4.3 rulename.type 6-22
6.2.4.4 rulename.match 6-23
6.2.4.5 rulename.subrules 6-23
6.2.4.6 txrule.timestamp 6-23
6.2.4.7 txrule.timestamp.format 6-24
6.2.4.8 txrule.seqid 6-24
6.2.4.9 txrule.txid 6-24
6.2.4.10 txrule.txowner 6-25
6.2.4.11 txrule.txname 6-25
6.2.4.12 oprule.timestamp 6-25
6.2.4.13 oprule.timestamp.format 6-25
6.2.4.14 oprule.seqid 6-26
6.2.4.15 oprule.txid 6-26
6.2.4.16 oprule.txowner 6-26
6.2.4.17 oprule.txname 6-26

6.2.4.18 oprule.schemandtable 6-27
6.2.4.19 oprule.schema 6-27
6.2.4.20 oprule.table 6-27
6.2.4.21 oprule.optype 6-27
6.2.4.22 oprule.optype.insertval 6-28
6.2.4.23 oprule.optype.updateval 6-28
6.2.4.24 oprule.optype.deleteval 6-28
6.2.4.25 oprule.txind 6-28
6.2.4.26 oprule.txind.beginval 6-28
6.2.4.27 oprule.txind.middleval 6-29
6.2.4.28 oprule.txind.endval 6-29
6.2.4.29 oprule.txind.wholeval 6-29
6.2.4.30 colrule.name 6-29
6.2.4.31 colrule.index 6-29
6.2.4.32 colrule.value 6-30
6.2.4.33 colrule.isnull 6-30
6.2.4.34 colrule.ismissing 6-30
6.2.4.35 colrule.before.value 6-30
6.2.4.36 colrule.before.isnull 6-30
6.2.4.37 colrule.before.ismissing 6-31
6.2.4.38 colrule.after.value 6-31
6.2.4.39 colrule.after.isnull 6-31
6.2.4.40 colrule.after.ismissing 6-31

Part III Oracle GoldenGate Java Delivery

7 Configuring Java Delivery

7.1 Configuring the JRE in the Properties File 7-1
7.2 Configuring Oracle GoldenGate for Java Delivery 7-1
7.2.1 Configuring a Replicat for Java Delivery 7-2
7.3 Configuring the Java Handlers 7-3

8 Running Java Delivery

8.1 Starting the Application 8-1
8.1.1 Starting Using Replicat 8-1
8.2 Restarting the Java Delivery 8-1
8.2.1 Restarting Java Delivery in Replicat 8-2

9 Configuring Event Handlers

9.1 Specifying Event Handlers 9-1
9.2 JMS Handler 9-2
9.3 File Handler 9-3
9.4 Custom Handlers 9-3
9.5 Formatting the Output 9-3
9.6 Reporting 9-4

10 Java Delivery Properties

10.1 Common Properties 10-1
10.1.1 Logging Properties 10-1
10.1.1.1 gg.log 10-1
10.1.1.2 gg.log.level 10-2
10.1.1.3 gg.log.file 10-2
10.1.1.4 gg.log.classpath 10-2
10.1.2 JVM Boot Options 10-2
10.1.2.1 jvm.bootoptions 10-2
10.2 Delivery Properties 10-3
10.2.1 General Properties 10-3
10.2.1.1 goldengate.userexit.writers 10-3
10.2.1.2 goldengate.userexit.chkptprefix 10-4
10.2.1.3 goldengate.userexit.nochkpt 10-4
10.2.1.4 goldengate.userexit.usetargetcols 10-4
10.2.2 Statistics and Reporting 10-4
10.2.2.1 jvm.stats.display 10-4
10.2.2.2 jvm.stats.full 10-5
10.2.2.3 jvm.stats.time | jvm.stats.numrecs 10-5
10.3 Java Application Properties 10-5
10.3.1 Properties for All Handlers 10-5
10.3.1.1 gg.handlerlist 10-6
10.3.1.2 gg.handler.name.type 10-6
10.3.2 Properties for Formatted Output 10-6
10.3.2.1 gg.handler.name.format 10-7
10.3.2.2 gg.handler.name.includeTables 10-7
10.3.2.3 gg.handler.name.excludeTables 10-8
10.3.2.4 gg.handler.name.mode, gg.handler.name.format.mode 10-8
10.3.3 Properties for CSV and Fixed Format Output 10-8
10.3.3.1 gg.handler.name.format.delim 10-9
10.3.3.2 gg.handler.name.format.quote 10-9
10.3.3.3 gg.handler.name.format.metacols 10-9

10.3.3.4 gg.handler.name.format.missingColumnChar 10-10
10.3.3.5 gg.handler.name.format.presentColumnChar 10-10
10.3.3.6 gg.handler.name.format.nullColumnChar 10-10
10.3.3.7 gg.handler.name.format.beginTxChar 10-10
10.3.3.8 gg.handler.name.format.middleTxChar 10-10
10.3.3.9 gg.handler.name.format.endTxChar 10-10
10.3.3.10 gg.handler.name.format.wholeTxChar 10-11
10.3.3.11 gg.handler.name.format.insertChar 10-11
10.3.3.12 gg.handler.name.format.updateChar 10-11
10.3.3.13 gg.handler.name.format.deleteChar 10-11
10.3.3.14 gg.handler.name.format.truncateChar 10-11
10.3.3.15 gg.handler.name.format.endOfLine 10-11
10.3.3.16 gg.handler.name.format.justify 10-11
10.3.3.17 gg.handler.name.format.includeBefores 10-12
10.3.4 File Writer Properties 10-12
10.3.4.1 gg.handler.name.file 10-12
10.3.4.2 gg.handler.name.append 10-12
10.3.4.3 gg.handler.name.rolloverSize 10-12
10.3.5 JMS Handler Properties 10-12
10.3.5.1 Standard JMS Settings 10-13
10.3.5.2 Group Transaction Properties 10-16
10.3.6 JNDI Properties 10-16
10.3.7 General Properties 10-16
10.3.7.1 gg.classpath 10-16
10.3.7.2 gg.report.time 10-17
10.3.7.3 gg.binaryencoding 10-17
10.3.8 Java Delivery Transaction Grouping 10-17

11 Developing Custom Filters, Formatters, and Handlers

11.1 Filtering Events 11-1
11.2 Custom Formatting 11-1
11.2.1 Coding a Custom Formatter in Java 11-1
11.2.2 Using a Velocity Template 11-3
11.3 Coding a Custom Handler in Java 11-4
11.4 Additional Resources 11-6

Part IV Troubleshooting the Oracle GoldenGate Adapters

12 Troubleshooting the Java Adapters

12.1 Checking for Errors 12-1
12.2 Reporting Issues 12-2

A List of Included Examples

B Customizing Logging

Preface

This guide contains information about configuring, and running Oracle GoldenGate for Big Data to extend the capabilities of Oracle GoldenGate instances.

- Audience (page xiii)
- Documentation Accessibility (page xiii)
- Related Information (page xiii)
- Conventions (page xiv)

Audience

This guide is intended for system administrators who are configuring and running Oracle GoldenGate for Big Data.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Accessible Access to Oracle Support

Oracle customers who have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Information

The Oracle GoldenGate Product Documentation Libraries are found at

Oracle GoldenGate

Oracle GoldenGate Application Adapters

Oracle GoldenGate for Big Data

Oracle GoldenGate Plug-in for EMCC

Oracle GoldenGate Monitor

Oracle GoldenGate for HP NonStop (Guardian)

Oracle GoldenGate Veridata

Oracle GoldenGate Studio

Additional Oracle GoldenGate information, including best practices, articles, and solutions, is found at:

Oracle GoldenGate A-Team Chronicles

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

Understanding Oracle GoldenGate for Big Data

This part of the book describes the concepts and basic structure of the Oracle GoldenGate for Big Data.

Topics:

- Understanding the Java Adapter and Oracle GoldenGate for Big Data (page 1-1)
- Introducing the Java Adapter (page 2-1)
- Configuring Logging (page 3-1)

1

Understanding the Java Adapter and Oracle GoldenGate for Big Data

This chapter includes the following sections.

Topics:

- Overview (page 1-1)
- Using Oracle GoldenGate Java Adapter Properties (page 1-3)
- Oracle GoldenGate Documentation (page 1-5)

1.1 Overview

This section provides an overview of the Oracle GoldenGate Adapters.

- Oracle GoldenGate Features (page 1-1)
- Adapter Integration Options (page 1-1)

1.1.1 Oracle GoldenGate Features

The Oracle GoldenGate Java Adapter integrates with Oracle GoldenGate instances.

The Oracle GoldenGate product enables you to:

- Captures transactional changes from a source database
- 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, platforms, and operating systems.

1.1.2 Adapter Integration Options

There are two major products which are based on the Oracle GoldenGate Adapter architecture:

- The Oracle GoldenGate Java Adapter is the overall framework. This product allows you to implement custom code to handle Oracle GoldenGate trail records according to their specific requirements. It comes built-in with Oracle GoldenGate File Writer module that can be used for flat file integration purposes.
- Oracle GoldenGate for Big Data. The Oracle GoldenGate for Big Data product contains built-in support to write operation data from Oracle GoldenGate trail records into various Big Data targets (such as, HDFS, HBase, Kafka, Flume, JDBC, Cassandra, and MongoDB). You do not need to write custom code to

integrate with Big Data applications. The functionality is separated into handlers that integrate with third party applications and formatters, which transform the data into various formats, such as Avro, JSON, delimited text, and XML. In certain instances, the integration to a third-party tool is proprietary, like the HBase API. In these instances, the formatter exists without an associated handler.

The Oracle GoldenGate Java Adapter and the Oracle GoldenGate for Big Data products have some crossover in functionality so the handler exists without an associated formatter. The following list details the major areas of functionality and in which product or products the functionality is included:

- Read JMS messages and deliver them as an Oracle GoldenGate trail. This feature is included in both Oracle GoldenGate Adapters and Oracle GoldenGate for Big Data products.
- Read an Oracle GoldenGate trail and deliver transactions to a JMS provider or other messaging system or custom application. This feature is included in both Oracle GoldenGate Adapters and Oracle GoldenGate for Big Data products.
- Read an Oracle GoldenGate trail and write transactions to a file that can be used by other applications. This feature is only included in the Oracle GoldenGate Adapters product.
- Read an Oracle GoldenGate trail and write transactions to a Big Data targets. The Big Data integration features are only included in the Oracle GoldenGate for Big Data product.
- Capturing Transactions to a Trail (page 1-2)
- Applying Transactions from a Trail (page 1-2)

1.1.2.1 Capturing Transactions to a Trail

Oracle GoldenGate message capture 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 processing 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 Oracle GoldenGate trail.

Currently this adapter supports capturing JMS text messages.

1.1.2.2 Applying Transactions from a Trail

Oracle GoldenGate Java Adapter delivery can be used to apply transactional changes to targets other than a relational database: for example, ETL tools (DataStage, Ab Initio, Informatica), JMS messaging, Big Data Applications, 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 File Writer 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 (for example, in XML) to JMS. The JMS provider is configurable to work with multiple JMS implementation; examples include ActiveMQ, JBoss Messaging, TIBCO, Oracle WebLogic JMS, WebSphere MQ, and others.
- Java API: custom 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.
- Big Data integration: writing transaction data from the source trail files into various Big Data targets can be achieved by means of setting configuration properties. The Oracle GoldenGate for Big Data product contains built in Big Data handlers to write to HDFS, HBase, Kafka, and Flume targets.

All four options have been implemented as extensions to the core Oracle GoldenGate product.

- For Java integration using either JMS or the Java API, use Oracle GoldenGate for Java.
- For Big Data integration, you can configure Oracle GoldenGate Replicat to integrate with the Oracle GoldenGate Big Data module. Writing to Big Data targets in various formats can be configured using a set of properties with no programming required.

1.2 Using Oracle GoldenGate Java Adapter Properties

The Oracle GoldenGate Java Adapters, Big Data Handlers, and formatters are configured through predefined properties. These properties are stored in a separate properties file called the Adapter Properties file. Oracle GoldenGate functionality requires that the Replicat process configuration files must be in the `dirprm` directory and that configuration files must adhere to the following naming conventions:

```
Replicat process name.prm
```

It is considered to be a best practice that the Adapter Properties files are also located in the `dirprm` directory and that the Adapter Properties files adhere to one of the following naming conventions:

```
Replicat process name.props
```

or

```
Replicat process name.properties
```

- Values in Property Files (page 1-3)
- Location of Property Files (page 1-4)
- Using Comments in the Property File (page 1-4)
- Variables in Property Names (page 1-4)

1.2.1 Values in Property Files

All properties in Oracle GoldenGate Adapter property files are of the form:

```
property.name=value
```

1.2.2 Location of Property Files

Sample Oracle GoldenGate Adapter properties files are installed to the `AdapterExamples` subdirectory of the installation directory. These files should be copied, renamed, and the contents modified as needed and then moved to the `dirprm` subdirectory.

You must specify each of these property files through parameters or environmental variables as explained below. These settings allow you to change the name or location, but it is recommended that you follow the best practice for naming and location.

The following sample files are included:

- `ffwriter.properties`

  This stores the properties for the file writer. It is set with the `CUSEREXIT` parameter.

- `jmsvam.properties`

  This stores properties for the JMS message capture VAM. This is set with the Extract `VAM` parameter.

- `javaue.properties`

  This stores properties for the Java application used for message delivery. It is set through the environmental variable:

  The name and location of the Adapter properties file is resolved by configuration in the Replicat process properties file.

  The following explains how to resolve the name and location of the Adapter Properties file in the Replicat process.

  ```
  TARGETDB LIBFILE libggjava.so SET property=dirprm/javaue.properties
  ```

1.2.3 Using Comments in the Property File

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. This allows testing configurations without losing previous property settings.

1.2.4 Variables in Property Names

Some properties have a variable in the property name. This allows identification of properties that are to be applied only in certain instances.

For example, you can declare more than one file writer using `goldengate.flatfilewriter.writers property` and then use the name of the file writer to set the properties differently:

1. Declare two file writers named `writer` and `writer2`:

   ```
   goldengate.flatfilewriter.writers=writer,writer2
   ```

2. Specify the properties for each of the file writers:

```
writer.mode=dsv
writer.files.onepertable=true
writer2.mode=ldv
writer2.files.onpertable=false
```

1.3 Oracle GoldenGate Documentation

For information on installing and configuring the core Oracle GoldenGate software for use with the Oracle GoldenGate adapter products, 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.
- *Administering Oracle GoldenGate*: Explains how to plan for, configure, and implement the Oracle GoldenGate replication solution on the Windows and UNIX platforms.
- *Reference for Oracle GoldenGate*: Contains detailed information about Oracle GoldenGate parameters, commands, and functions for the Windows and UNIX platforms.

2
Introducing the Java Adapter

This chapter includes the following sections.

Topics:

- Oracle GoldenGate VAM Message Capture (page 2-1)
- Oracle GoldenGate Java Delivery (page 2-2)
- Delivery Configuration Options (page 2-3)

2.1 Oracle GoldenGate VAM Message Capture

Oracle GoldenGate VAM Message Capture only works with the Oracle GoldenGate Extract process. Oracle GoldenGate message capture connects to JMS messaging to parse messages and send them through a VAM interface to an Oracle GoldenGate Extract process 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. Java 8 is a required dependency for Oracle GoldenGate VAM Message Capture.

Using Oracle GoldenGate JMS message capture requires the dynamically linked shared VAM library that is attached to the Oracle GoldenGate Extract process.

In a situation where a trail definition file is required to parse the resulting trail written by the Extract process, a separate utility, `gendef`, can be used to create an Oracle GoldenGate source definitions file.

- Message Capture Configuration Options (page 2-1)
- Typical Configuration (page 2-1)

2.1.1 Message Capture Configuration Options

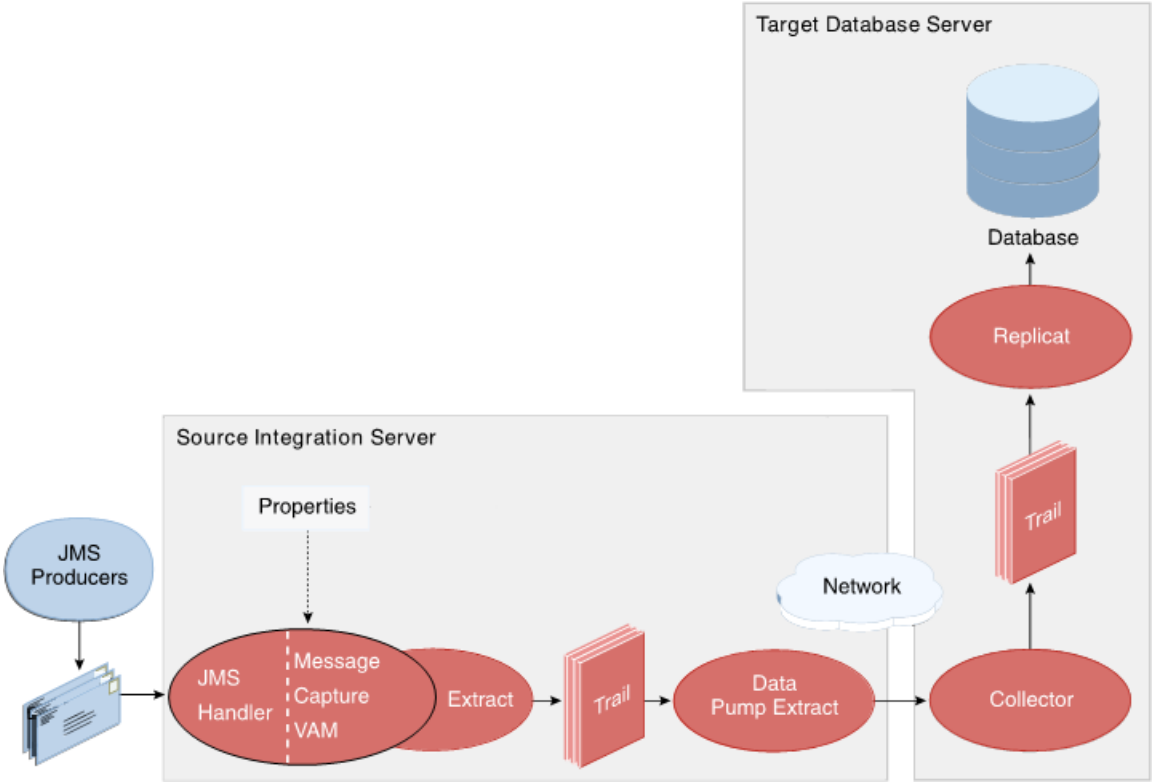
The options for configuring the three parts of message capture are:

- 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.

2.1.2 Typical Configuration

The following diagram shows a typical configuration for capturing JMS messages.

Figure 2-1 Configuration for JMS Message Capture



In this configuration, JMS messages are picked up by the Oracle GoldenGate Adapter JMS Handler and transferred using the adapter's message capture VAM to an Extract process. The Extract writes the data to a trail which is sent over the network by a Data Pump Extract to an Oracle GoldenGate target instance. The target Replicat then uses the trail to update the target database.

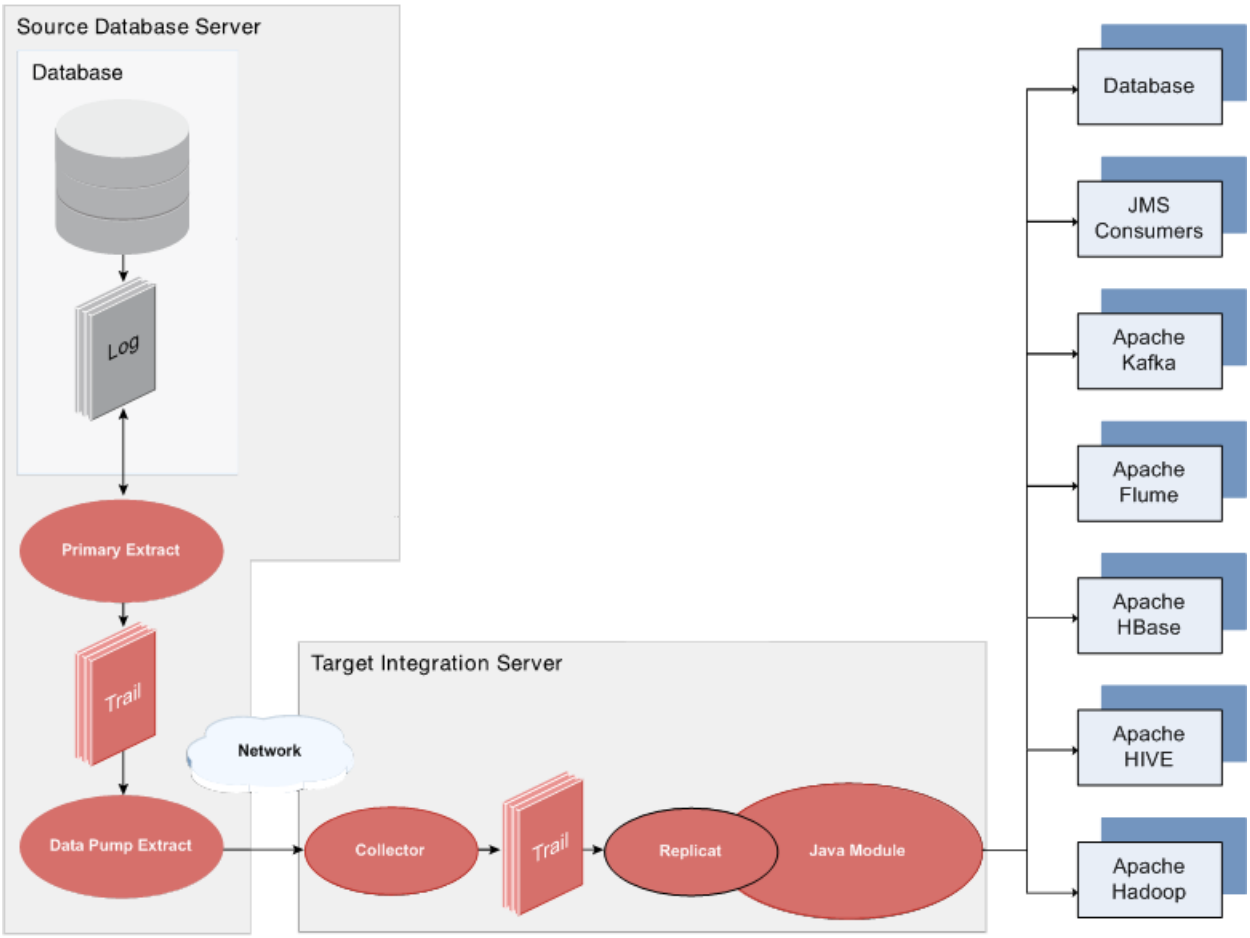
2.2 Oracle GoldenGate Java Delivery

Through the Oracle GoldenGate Java API, transactional data captured by Oracle GoldenGate can be delivered to targets other than a relational database, such as a JMS (Java Message Service), files written to disk, streaming data to a Big Data application, or integration with a custom application Java API. Oracle GoldenGate Java Delivery can work with either an Extract or Replicat process. Using the Oracle GoldenGate Replicat process is considered the best practice. Oracle GoldenGate Java Delivery requires Java 8 as a dependency.

Oracle GoldenGate for Java provides the ability to execute Java code from the Oracle GoldenGate Replicat process. Using Oracle GoldenGate for Java requires the following conditions to be met:

- A dynamically linked or shared library, implemented in C/C++, integrating an extension module of Oracle GoldenGate Replicat process.
- A set of Java libraries (JARs), which comprise the Oracle GoldenGate Java API. This Java framework communicates with the Replicat through the Java Native Interface (JNI).
- Java 8 must be installed and accessible on the machine hosting the Oracle GoldenGate Java Delivery process or processes. Environmental variables must be correctly set to resolve Java and its associated libraries.

Figure 2-2 Configuration for Delivering JMS Messages



2.3 Delivery Configuration Options

The Java delivery module is loaded by the GoldenGate Replicat process, which is configured using the Replicat parameter file. Upon loading, the Java Delivery module is subsequently configured based on the configuration present in the Adapter Properties file. Application behavior can be customized by:

- Editing the property files; for example to:
  - Set target types, host names, port numbers, output file names, JMS connection settings;
  - Turn on/off debug-level logging, and so on.
  - Identify which message format should be used.
- Records can be custom formatted by:
  - Setting properties for the pre-existing format process (for fixed-length or field-delimited message formats, XML, JSON, or Avro formats);
  - Customizing message templates, using the Velocity template macro language;
  - (Optional) Writing custom Java code.

- (Optional) Writing custom Java code to provide custom handling of transactions and operations, do filtering, or implementing custom message formats.

There are existing implementations (handlers) for sending messages using JMS and for writing out files to disk. For Big Data targets, there are built in integration handlers to write to supported databases.

There are several predefined message formats for sending the messages (for example, XML or field-delimited); or custom formats can be implemented using templates. Each handler has documentation that describes its configuration properties; for example, a file name 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.

3
Configuring Logging

This chapter includes the following sections.

Topics:

- Oracle GoldenGate Java Adapter Default Logging (page 3-1)
- Recommended Logging Settings (page 3-1)

3.1 Oracle GoldenGate Java Adapter Default Logging

Logging is set up by default for the Oracle GoldenGate Adapters.

- Default Implementation Type (page 3-1)
- Default Message Logging (page 3-1)
- Log File Name (page 3-1)
- Logging Problems (page 3-1)

3.1.1 Default Implementation Type

The default type of implementation for the Oracle GoldenGate Adapters is the `jdk` option. This is the built-in Java logging called `java.util.logging`.

3.1.2 Default Message Logging

The default log file is created in the standard report directory. It is named for the associated Replicat process. Problems are logged to the report file and the log file.

3.1.3 Log File Name

By default log files are written to the *`installation_directory`*`/dirrpt` directory. The name of the log file includes the Replicat *`group_name`* and it has an extension of `log`.

3.1.4 Logging Problems

An overview of a problem is written to the Replicat Report file and the details of the problem are written to the log file.

3.2 Recommended Logging Settings

Oracle recommends that you use log4j logging instead of the JDK default for unified logging for the Java user exit. Using log4j provides unified logging for the Java module when running with the Oracle GoldenGate Replicat process.

- Changing to the Recommended Logging Type (page 3-2)

3.2.1 Changing to the Recommended Logging Type

To change the recommended log4j logging implementation, add the configuration shown in the following example to the Java Adapter Properties file.

```
gg.log=log4j
gg.log.level=info
```

The `gg.log` level can be set to `none`, `error`, `warn`, `info`, `debug`, or `trace`. The default log level is `info`. The debug and trace log levels are only recommended for troubleshooting as these settings can adversely affect performance.

The result is that a log file for the Java module will be created in the `dirrpt` directory with the following naming convention:

```
<process name>_<log level>log4j.log
```

Therefore if the Oracle GoldenGate Replicat process is called `javaue`, and the `gg.log.level` is set to debug, the resulting log file name is:

```
javaue_debug_log4j.log
```

Part II
Capturing JMS Messages

This part of the book explains using the Oracle GoldenGate Adapter to capture Java Message Service (JMS) messages to be written to an Oracle GoldenGate trail.

Topics:

- Configuring Message Capture (page 4-1)
- Parsing the Message (page 5-1)
- Message Capture Properties (page 6-1)

4
Configuring Message Capture

This chapter explains how to configure the VAM Extract to capture JMS messages.
Topics:

- Configuring the VAM Extract (page 4-1)
- Connecting and Retrieving the Messages (page 4-2)

4.1 Configuring the VAM Extract

JMS Capture only works with the Oracle GoldenGate Extract process. To run the Java message capture application you need the following:

- Oracle GoldenGate for Java Adapter
- Extract process
- Extract parameter file configured for message capture
- Description of the incoming data format, such as a source definitions file.
- Java 8 installed on the host machine

- Adding the Extract (page 4-1)
- Configuring the Extract Parameters (page 4-2)
- Configuring Message Capture (page 4-2)

4.1.1 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.

4.1.2 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.
`TRANLOGOPTIONS VAMCOMPATIBILITY 1`	Specifies the original (1) implementation of the VAM is to be used.
`TRANLOGOPTIONS GETMETADATAFROMVAM`	Specifies that metadata will be sent by the VAM.
`EXTTRAIL dirdat/id`	Specifies the identifier of the target trail Extract creates.

4.1.3 Configuring Message Capture

Message capture is configured by the properties in the VAM properties file (Adapter 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.

4.2 Connecting and Retrieving the Messages

To process JMS messages you must configure the connection to the JMS interface, retrieve and parse the messages in a transaction, write each message to a trail, commit the transaction, and remove its messages from the queue.

- Connecting to JMS (page 4-2)
- Retrieving Messages (page 4-3)
- Completing the Transaction (page 4-3)

4.2.1 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.

4.2.2 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.

4.2.3 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.

5
Parsing the Message

This chapter includes the following sections.

Topics:

- Parsing Overview (page 5-1)
- Fixed Width Parsing (page 5-5)
- Delimited Parsing (page 5-9)
- XML Parsing (page 5-11)
- Source Definitions Generation Utility (page 5-19)

5.1 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 must 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 (page 5-1)
- Source and Target Data Definitions (page 5-2)
- Required Data (page 5-2)
- Optional Data (page 5-4)

5.1.1 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.

5.1.2 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 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.

5.1.3 Required Data

The following information is required for the parsers to translate the messages:

- Transaction Identifier (page 5-2)
- Sequence Identifier (page 5-2)
- Timestamp (page 5-3)
- Table Name (page 5-3)
- Operation Type (page 5-3)
- Column Data (page 5-4)

5.1.3.1 Transaction Identifier

The transaction identifier (`txid`) groups operations into transactions as they are 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.

5.1.3.2 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 (for example, 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.

5.1.3.3 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.

5.1.3.4 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.

5.1.3.5 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.

5.1.3.6 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.

5.1.4 Optional Data

The following data may be included, but is not required.

- Transaction Indicator (page 5-4)
- Transaction Name (page 5-4)
- Transaction Owner (page 5-4)

5.1.4.1 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.

5.1.4.2 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.

5.1.4.3 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.

5.2 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 (page 5-7).

- Header (page 5-5)
- Header and Record Data Type Translation (page 5-7)
- Key identification (page 5-7)
- Using a Source Definition File (page 5-7)

5.2.1 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.

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

Example 5-1 Specifying a Header

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

For the preceding example, you must set the following properties:

```
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 (page 5-6)
- Specifying timestamp Formats (page 5-6)
- Specifying the Function (page 5-6)

5.2.1.1 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
```

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

Example 5-2 Specifying Compound Table Names

```
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).
```

For the preceding example, you must set the following properties:

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

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

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

5.2.1.2 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.

Example 5-3 Specifying timestamp formats

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

5.2.1.3 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`.

Example 5-4 Specifying the Function

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

5.2.2 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 data type.
- A `PIC X` field is translated to the `CHAR` data type of the indicated size.
- A `PIC 9` field is translated to a `NUMBER` data type 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)`

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.

5.2.3 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)
```

5.2.4 Using a Source Definition File

You can use fixed width parsing based on a data definition that comes from an Oracle GoldenGate source definition file. This is similar to Cobol copybook because a source definition file contains the position and the length of each field of participating tables. To use a source definition file, you must set the following properties:

```
fixed.userdefs.tables=qasource.HEADER
fixed.userdefs.qasource.HEADER.columns=optype,schemaandtable
fixed.userdefs.qasource.HEADER.optype=vchar 3
fixed.userdefs.qasource.HEADER.schemaandtable=vchar 30

fixed.header=qasource.HEADER
```

The following example defines a header section of a total length of 33 characters; the first 3 characters are the operation type, and the last 30 characters is the table name. The layout of all records to be parsed must start with the complete header section as defined in the `fixed.userdefs` properties. For each record, the header section is immediately followed by the content of all column data for the corresponding table. The column data must be strictly laid out according to its offset and length defined in the source definition file. Specifically, the offset information is the fourth field (Fetch Offset) of the column definition and the length information is the third field (External Length) of the column definition. The following is an example of a definition for GG.JMSCAP_TCUSTMER:

```
Definition for table GG.JMSCAP_TCUSTMER
Record length: 78
Syskey: 0
Columns: 4
CUST_CODE   64      4        0  0  0 1 0      4      4      0 0 0 0 0 1    0 1 0
NAME        64     30       10  0  0 1 0     30     30      0 0 0 0 0 1    0 0 0
CITY        64     20       46  0  0 1 0     20     20      0 0 0 0 0 1    0 0 0
STATE        0      2       72  0  0 1 0      2      2      0 0 0 0 0 1    0 0 0
End of definition
```

The fixed width data for GG.JMSCAP_TCUSTMER may be similar to the following where the offset guides have been added to each section for clarity:

```
0         1         2         3  0         1         2         3         4
5         6         7         8
012345678901234567890123456789012012345678901234567890123456789012345678901234567890123
234567890123456789012345678901234567890
I  GG.JMSCAP_TCUSTMER            WILL      BG SOFTWARE CO.
SEATTLE                   WA
I  GG.JMSCAP_TCUSTMER            JANE      ROCKY FLYER INC.
DENVER                    CO
I  GG.JMSCAP_TCUSTMER            DAVE      DAVE'S PLANES INC.
TALLAHASSEE               FL
I  GG.JMSCAP_TCUSTMER            BILL      BILL'S USED CARS
DENVER                    CO
I  GG.JMSCAP_TCUSTMER            ANN       ANN'S BOATS
SEATTLE                   WA
U  GG.JMSCAP_TCUSTMER            ANN       ANN'S BOATS                         NEW
YORK                  NY
```

You can choose to specify shorter data records, which means that only some of the earlier columns are present. To do this, the following requirements must be met:

- None of the missing or omitted columns are part of the key and
- all columns that are present contain complete data according to their respective External Length information

.

5.3 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 data type of the columns in the tables.

The format of the delimited message is:

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

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`.

The message to be parsed *must* contain at least the header and metadata columns. If the number of columns is fewer than the number of header and meta columns, then the capture process terminates and provides an error message.

The remaining number of columns after the header and metadata columns are the column data for the corresponding table, specified in the order of the columns in the resolved metadata. Ideally, the number of table columns present in the message is exactly the same as the expected number of columns according to the metadata. However, missing columns in the message towards the end of message is allowed and the parser marks those last columns (not present in the rest of the message) as missing column data.

Although missing data is allowed from parser perspective, if the key @ column(s) is/are missing, then the capture process will also terminate.

Oracle GoldenGate primary key updates and unified updates are not supported. The only supported operations are inserts, updates, deletes, and truncates.

- Metadata Columns (page 5-9)
- Parsing Properties (page 5-10)
- Parsing Steps (page 5-11)

5.3.1 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.

5.3.2 Parsing Properties

Properties can be set to describe delimiters, values, and date and time formats.

- Properties to Describe Delimiters (page 5-10)
- Properties to Describe Values (page 5-10)
- Properties to Describe Date and Time (page 5-11)

5.3.2.1 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.

5.3.2.2 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

5.3.2.3 Properties to Describe Date and Time

The default format `YYYY-DD-MM HH:MM:SS.FFF` is used to parse dates. You 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
```

5.3.3 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.

5.4 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 (page 5-11)
- XML Parsing Rules (page 5-12)
- XPath Expressions (page 5-13)
- Other Value Expressions (page 5-15)
- Transaction Rules (page 5-15)
- Operation Rules (page 5-16)
- Column Rules (page 5-17)
- Overall Rules Example (page 5-18)

5.4.1 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.

Example 5-5 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.

Example 5-6 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.

5.4.2 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
```

5.4.3 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: (page 5-13)
- Supported Expressions (page 5-14)
- Obtaining Data Values (page 5-14)

5.4.3.1 Supported Constructs:

Supported Constructs	Description
`/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.

Supported Constructs	Description
`[x=v]`	Match when x is equal to some value v where x can be: • `@att` - some attribute value • `text()` - some text value • `name()` - some name value • `position()` - the element position

5.4.3.2 Supported Expressions

Supported Expressions	Descriptions
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']`

5.4.3.3 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 in the preceding table, but must end with one of the value accessors listed below.

Value Accessors	Description
`@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.

Value Accessors	Description
`position()`	The position of an element in its parent.

Example 5-7 Examples of Extracting Data Values

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.

5.4.4 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"
```

5.4.5 Transaction Rules

The rule that specifies the boundary for a transaction is at the highest level. Messages may contain a single transaction, multiple transactions, or a part of a transaction that spans 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.

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

Example 5-8 JMS Timestamp

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

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

Example 5-9 ts Timestamp

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

5.4.6 Operation Rules

An operation rule can either be a sub rule 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 if 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 property is optional and not valid if the operation rule is a sub rule of a transaction rule.

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

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

Example 5-10 Operation

```
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.

Example 5-11 Operation example

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

5.4.7 Column Rules

A column rule must be a sub rule 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 `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.

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

Example 5-12 Dynamic Extraction of Column Information

```
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`.

Example 5-13 Static Matching of Elements to Columns

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

5.4.8 Overall Rules Example

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

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

Dynamic	Static

`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`
	`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`

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

5.5 Source Definitions Generation Utility

By default, the JMS capture process writes metadata information in the produced trail files, allowing trail file consumers to understand the structure of the trail records without any help from an external definition file.

In situations where the metadata in trail feature is disabled, the trail file consumers will still need the definition file to correctly parse the trail records. For this purpose 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.

6
Message Capture Properties

This chapter includes the following sections.

Topics:

- Logging and Connection Properties (page 6-1)
- Parser Properties (page 6-5)

6.1 Logging and Connection Properties

The following properties control the connection to JMS and the log file names, error handling, and message output.

- Logging Properties (page 6-1)
- JMS Connection Properties (page 6-2)
- JNDI Properties (page 6-5)

6.1.1 Logging Properties

Logging is controlled by the following properties.

- gg.log (page 6-1)
- gg.log.level (page 6-1)
- gg.log.file (page 6-2)
- gg.log.classpath (page 6-2)

6.1.1.1 gg.log

Specifies the type of logging that is to be used. The default implementation is the `JDK` option. This is the built-in Java logging called `java.util.logging` (`JUL`). The other logging options are `log4j` or `logback`. The syntax is:

```
gg.log={JDK|log4j|logback}
```

For example, to set the type of logging to `log4j`:

```
gg.log=log4j
```

The log file is created in the report subdirectory of the installation. The default log file name includes the group name of the associated Extract and the file extension is `log`.

6.1.1.2 gg.log.level

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

```
gg.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
gg.log.level=INFO
```

6.1.1.3 gg.log.file

Specifies the path to the log file. The syntax is:

```
gg.log.file=path_to_file
```

Where the *`path_to_file`* is the fully defined location of the log file. This allows a change to the name of the log, but you must include the Replicat name if you have more than one Replicat to avoid one overwriting the log of the other.

6.1.1.4 gg.log.classpath

Specifies the classpath to the JARs used to implement logging.

```
gg.log.classpath=path_to_jars
```

6.1.2 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 (page 6-2)
- jms.report.output (page 6-3)
- jms.report.time (page 6-3)
- jms.report.records (page 6-3)
- jms.id (page 6-4)
- jms.destination (page 6-4)
- jms.connectionFactory (page 6-4)
- jms.user, jms.password (page 6-4)

6.1.2.1 jvm.boot options

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

The syntax is:

```
jvm.bootoptions=option[, 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 maximum 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).

6.1.2.2 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.

6.1.2.3 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
```

6.1.2.4 jms.report.records

Specifies the frequency of report generation based on number of records. The syntax is:

```
jms.report.records=number
```

The following example writes a report every 1000 records.

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

6.1.2.5 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 your 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.

6.1.2.6 jms.destination

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

```
jms.destination=jndi_name
```

For example:

```
jms.destination=sampleQ
```

6.1.2.7 jms.connectionFactory

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

```
jms.connectionFactory=jndi_name
```

For example

```
jms.connectionFactory=ConnectionFactory
```

6.1.2.8 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
```

6.1.3 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
```

6.2 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 (page 6-5)
- Fixed Parser Properties (page 6-6)
- Delimited Parser Properties (page 6-11)
- XML Parser Properties (page 6-21)

6.2.1 Setting the Type of Parser

The following property sets the parser type.

- parser.type (page 6-5)

6.2.1.1 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
```

6.2.2 Fixed Parser Properties

The following properties are required for the fixed parser.

- fixed.schematype (page 6-6)
- fixed.sourcedefs (page 6-7)
- fixed.copybook (page 6-7)
- fixed.header (page 6-7)
- fixed.seqid (page 6-7)
- fixed.timestamp (page 6-8)
- fixed.timestamp.format (page 6-8)
- fixed.txid (page 6-8)
- fixed.txowner (page 6-9)
- fixed.txname (page 6-9)
- fixed.optype (page 6-9)
- fixed.optype.insertval (page 6-9)
- fixed.optype.updateval (page 6-9)
- fixed.optype.deleteval (page 6-10)
- fixed.table (page 6-10)
- fixed.schema (page 6-10)
- fixed.txind (page 6-10)
- fixed.txind.beginval (page 6-10)
- fixed.txind.middleval (page 6-11)
- fixed.txind.endval (page 6-11)
- fixed.txind.wholeval (page 6-11)

6.2.2.1 fixed.schematype

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

```
fixed.schematype={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.

6.2.2.2 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
```

6.2.2.3 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
```

6.2.2.4 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
```

6.2.2.5 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
```

6.2.2.6 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
```

6.2.2.7 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
```

6.2.2.8 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
```

6.2.2.9 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"
```

6.2.2.10 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"
```

6.2.2.11 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
```

6.2.2.12 fixed.optype.insertval

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

```
fixed.optype.insertval={value|\xhex_value}
```

For example:

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

6.2.2.13 fixed.optype.updateval

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

```
fixed.optype.updateval={value|\xhex_value}
```

For example:

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

6.2.2.14 fixed.optype.deleteval

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

```
fixed.optype.deleteval={value|\xhex_value}
```

For example:

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

6.2.2.15 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
```

6.2.2.16 fixed.schema

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

```
fixed.schema="value"
```

For example:

```
fixed.schema="OGG"
```

6.2.2.17 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
```

6.2.2.18 fixed.txind.beginval

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

```
fixed.txind.beginval={value|\xhex_value}
```

For example:

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

6.2.2.19 fixed.txind.middleval

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

```
fixed.txind.middleval={value|\xhex_value}
```

For example:

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

6.2.2.20 fixed.txind.endval

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

```
fixed.txind.endval={value|\xhex_value}
```

For example:

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

6.2.2.21 fixed.txind.wholeval

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

```
fixed.txind.wholeval={value|\xhex_value}
```

For example:

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

6.2.3 Delimited Parser Properties

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

- delim.sourcedefs (page 6-12)
- delim.header (page 6-12)
- delim.seqid (page 6-13)
- delim.timestamp (page 6-13)
- delim.timestamp.format (page 6-13)
- delim.txid (page 6-14)
- delim.txowner (page 6-14)
- delim.txname (page 6-14)
- delim.optype (page 6-14)
- delim.optype.insertval (page 6-14)
- delim.optype.updateval (page 6-15)
- delim.optype.deleteval (page 6-15)

- delim.schemaandtable (page 6-15)
- delim.schema (page 6-15)
- delim.table (page 6-15)
- delim.txind (page 6-16)
- delim.txind.beginval (page 6-16)
- delim.txind.middleval (page 6-16)
- delim.txind.endval (page 6-16)
- delim.txind.wholeval (page 6-16)
- delim.fielddelim (page 6-17)
- delim.linedelim (page 6-17)
- delim.quote (page 6-17)
- delim.nullindicator (page 6-17)
- delim.fielddelim.escaped (page 6-17)
- delim.linedelim.escaped (page 6-18)
- delim.quote.escaped (page 6-18)
- delim.nullindicator.escaped (page 6-19)
- delim.hasbefores (page 6-19)
- delim.hasnames (page 6-20)
- delim.afterfirst (page 6-20)
- delim.isgrouped (page 6-20)
- delim.dateformat | delim.dateformat.table | delim.dateform.table.column (page 6-20)

6.2.3.1 delim.sourcedefs

Specifies the location of the source definitions file to use.

```
delim.sourcedefs=file_location
```

For example:

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

6.2.3.2 delim.header

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

```
delim.header=name[,name2][. . .]
```

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
```

6.2.3.3 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
```

6.2.3.4 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
```

6.2.3.5 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
```

6.2.3.6 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
```

6.2.3.7 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"
```

6.2.3.8 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"
```

6.2.3.9 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
```

6.2.3.10 delim.optype.insertval

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

```
delim.optype.insertval={value|\xhex_value}
```

For example:

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

6.2.3.11 delim.optype.updateval

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

```
delim.optype.updateval={value|\xhex_value}
```

For example:

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

6.2.3.12 delim.optype.deleteval

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

```
delim.optype.deleteval={value|\xhex_value}
```

For example:

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

6.2.3.13 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
```

6.2.3.14 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"
```

6.2.3.15 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
```

6.2.3.16 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
```

6.2.3.17 delim.txind.beginval

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

```
delim.txind.beginval={value|\xhex_value}
```

For example:

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

6.2.3.18 delim.txind.middleval

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

```
delim.txind.middleval={value|\xhex_value}
```

For example:

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

6.2.3.19 delim.txind.endval

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

```
delim.txind.endval={value|\xhex_value}
```

For example:

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

6.2.3.20 delim.txind.wholeval

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

```
delim.txind.wholeval={value|\xhex_value}
```

For example:

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

6.2.3.21 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|\xhex_value}
```

For example:

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

6.2.3.22 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|\xhex_value}
```

For example:

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

6.2.3.23 delim.quote

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

```
delim.quote={value|\xhex_value}
```

For example:

```
delim.quote="
```

6.2.3.24 delim.nullindicator

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

```
delim.nullindicator={value|\xhex_value}
```

For example:

```
delim.nullindicator=NULL
```

6.2.3.25 delim.fielddelim.escaped

Specifies the value that will replace the field delimiter when the field delimiter occurs in the input field. The syntax is:

```
delim.fielddelim.escaped={value|\xhex_value}
```

For example, given the following property settings:

```
delim.fielddelim=-
delim.fielddelim.escaped=$#$
```

If the data does not contain the hyphen delimiter within any of the field values:

```
one two three four
```

The resulting delimited data is:

```
one-two-three-four
```

If there are hyphen (-) delimiters within the field values:

```
one two three four-fifths two-fifths
```

The resulting delimited data is:

```
one-two-three-four$#$fifths-two$#$fifths
```

6.2.3.26 delim.linedelim.escaped

Specifies the value that will replace the line delimiter when the line delimiter occurs in the input data. The syntax is:

```
delim.linedelim.escaped={value|\xhex_value}
```

For example, given the following property settings:

```
delim.linedelim=\
delim.linedelim.escaped=%/%
```

If the input lines are:

```
These are the lines and they
do not contain the delimiter.
```

Because the lines do not contain the backslash (\), the result is:

```
These are the lines and they\
do not contain the delimiter.\
```

However, if the input lines do contain the delimiter:

```
These are the lines\data values
and they do contain the delimiter.
```

So the results are:

```
These are the lines%/%data values\
and they do contain the delimiter.\
```

6.2.3.27 delim.quote.escaped

Specifies the value that will replace a quote delimiter when the quote delimiter occurs in the input data. The syntax is:

```
delim.quote.escaped={value|\xhex_value}
```

For example, given the following property settings:

```
delim.quote="
delim.quote.escaped="'"
```

If the input data does not contain the quote (") delimiter:

```
It was a very original play.
```

The result is:

```
"It was a very original play."
```

However, if the input data does contain the quote delimiter:

```
It was an "uber-original" play.
```

The result is:

```
"It was an "'"uber-original"'" play."
```

6.2.3.28 delim.nullindicator.escaped

Specifies the value that will replace a null indicator when a null indicator occurs in the input data. The syntax is:

```
delim.nullindicator.escaped={value|\xhex_value}
```

For example, given the following property settings:

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

When the input data does not contain a `NULL` value or a `NULL` indicator:

```
1 2 3 4 5
```

The result is

```
1,2,3,4,5
```

When the input data contains a `NULL` value:

```
1 2 4 5
```

The result is

```
1,2,NULL,4,5
```

When the input data contains a `NULL` indicator:

```
1 2 NULL 4 5
```

The result is:

```
1,2,$NULL$,4,5
```

6.2.3.29 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
```

6.2.3.30 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
```

6.2.3.31 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
```

6.2.3.32 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
```

6.2.3.33 delim.dateformat | delim.dateformat.*table* | delim.dateform.table.column

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

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

Where:

- *format* is the format defined for `parser.timestamp.format`.
- *table* is the fully qualified name of the table that is currently being processed.
- *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
```

6.2.4 XML Parser Properties

The following properties are used by the XML parser.

- xml.sourcedefs (page 6-22)
- xml.rules (page 6-22)
- rulename.type (page 6-22)
- rulename.match (page 6-23)
- rulename.subrules (page 6-23)
- txrule.timestamp (page 6-23)
- txrule.timestamp.format (page 6-24)
- txrule.seqid (page 6-24)
- txrule.txid (page 6-24)
- txrule.txowner (page 6-25)
- txrule.txname (page 6-25)
- oprule.timestamp (page 6-25)
- oprule.timestamp.format (page 6-25)
- oprule.seqid (page 6-26)
- oprule.txid (page 6-26)
- oprule.txowner (page 6-26)
- oprule.txname (page 6-26)
- oprule.schemandtable (page 6-27)
- oprule.schema (page 6-27)
- oprule.table (page 6-27)
- oprule.optype (page 6-27)
- oprule.optype.insertval (page 6-28)
- oprule.optype.updateval (page 6-28)
- oprule.optype.deleteval (page 6-28)

- oprule.txind (page 6-28)
- oprule.txind.beginval (page 6-28)
- oprule.txind.middleval (page 6-29)
- oprule.txind.endval (page 6-29)
- oprule.txind.wholeval (page 6-29)
- colrule.name (page 6-29)
- colrule.index (page 6-29)
- colrule.value (page 6-30)
- colrule.isnull (page 6-30)
- colrule.ismissing (page 6-30)
- colrule.before.value (page 6-30)
- colrule.before.isnull (page 6-30)
- colrule.before.ismissing (page 6-31)
- colrule.after.value (page 6-31)
- colrule.after.isnull (page 6-31)
- colrule.after.ismissing (page 6-31)

6.2.4.1 xml.sourcedefs

Specifies the location of the source definitions file.

```
xml.sourcedefs=file_location
```

For example:

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

6.2.4.2 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
```

6.2.4.3 *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
```

6.2.4.4 *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 sub rule of `stattxtule`, the following example activates the `statoprule` if the parent rule's matching nodes have child elements of `NewMyTableEntry`.

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

6.2.4.5 *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 sub-rules are determined by the parent type. Transaction rules can only have operation sub-rules. Operation rules can have operation or column sub-rules. Column rules cannot have sub-rules.

For example:

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

6.2.4.6 *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
```

6.2.4.7 *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
```

6.2.4.8 *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
```

6.2.4.9 *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
```

6.2.4.10 *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"
```

6.2.4.11 *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"
```

6.2.4.12 *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()
```

6.2.4.13 *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
```

6.2.4.14 *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.

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
```

6.2.4.15 *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.

```
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
```

6.2.4.16 *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"
```

6.2.4.17 *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"
```

6.2.4.18 *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"
```

6.2.4.19 *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
```

6.2.4.20 *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
```

6.2.4.21 *oprule*.optype

Specifies the XPath expression, JMS property or literal value that contains the `optype` to be validated against an `optype insertval`. 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"
```

6.2.4.22 *oprule*.optype.insertval

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

```
oprule.optype.insertval={value|\xhex_value}
```

For example:

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

6.2.4.23 *oprule*.optype.updateval

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

```
oprule.optype.updateval={value|\xhex_value}
```

For example:

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

6.2.4.24 *oprule*.optype.deleteval

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

```
oprule.optype.deleteval={value|\xhex_value}
```

For example:

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

6.2.4.25 *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
```

6.2.4.26 *oprule*.txind.beginval

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

```
oprule.txind.beginval={value|\xhex_value}
```

For example:

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

6.2.4.27 *oprule*.txind.middleval

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

```
oprule.txind.middleval={value|\xhex_value}
```

For example:

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

6.2.4.28 *oprule*.txind.endval

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

```
oprule.txind.endval={value|\xhex_value}
```

For example:

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

6.2.4.29 *oprule*.txind.wholeval

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

```
oprule.txind.wholeval={value|\xhex_value}
```

For example:

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

6.2.4.30 *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"
```

6.2.4.31 *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
```

6.2.4.32 *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()
```

6.2.4.33 *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
```

6.2.4.34 *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
```

6.2.4.35 *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()
```

6.2.4.36 *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
```

6.2.4.37 *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
```

6.2.4.38 *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()
```

6.2.4.39 *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
```

6.2.4.40 *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
```

Part III
Oracle GoldenGate Java Delivery

This part of the book contains information on using Oracle GoldenGate Adapters to process transaction records and apply it to various targets by means of Java module.

Introduction to GoldenGate for Big Data discusses how to use the Oracle GoldenGate Java Adapter to apply transaction records into various Big Data targets.

Topics:

- Configuring Java Delivery (page 7-1)
- Running Java Delivery (page 8-1)
- Configuring Event Handlers (page 9-1)
- Java Delivery Properties (page 10-1)
- Developing Custom Filters, Formatters, and Handlers (page 11-1)

7
Configuring Java Delivery

This chapter includes the following sections.

Topics:

- Configuring the JRE in the Properties File (page 7-1)
- Configuring Oracle GoldenGate for Java Delivery (page 7-1)
- Configuring the Java Handlers (page 7-3)

7.1 Configuring the JRE in the Properties File

The current release of Oracle GoldenGate Java Delivery requires Java 8. Refer to the section on configuring Java for how to correctly access Java and the required Java shared libraries. Modify the Adapter 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):

```
jvm.bootoptions=-Djava.class.path=.:ggjava/ggjava.jar -Xmx512m -Xmx64m
```

Note the following options in particular:

- `java.class.path` must include pathing to the core application (`ggjava/ggjava.jar`). The current directory (.) should be included as well in the classpath. Logging initializes when the JVM is loaded therefore the `java.class.path` variable should including any pathing to logging properties files (such as `log4j` properties files). The dependency JARs required for logging functionality are included in `ggjava.jar` and do not need to be explicitly included. Pathing can reference files and directories relative to the Oracle GoldenGate install directory, to allow storing Java property files, Velocity templates and other classpath resources in the `dirprm` subdirectory. It is also possible to append to the classpath in the Java application properties file. Pathing to handler dependency JARs can be added here as well. However, it is considered to be a better practice to use the `gg.classpath` variable to include any handler dependencies.
- The `jvm.bootoptions` property also allows you to control the initial heap size of the JVM (Xms) and the maximum heap size of the JVM (Xmx). Increasing the maximum heap size can improve performance by requiring less frequent garbage collections. Additionally, you may need to increase the maximum heap size if a Java out of memory exception occurs.

Once the properties file is correctly configured for your system, it usually remains unchanged. See Common Properties (page 10-1), for additional configuration options.

7.2 Configuring Oracle GoldenGate for Java Delivery

Java Delivery is compatible with the Oracle GoldenGate Replicat process. Transaction data is read from the Oracle GoldenGate trail files and delivered to the Oracle

GoldenGate Java Delivery module across JNI interface. The data is transferred to the Oracle GoldenGate Java Delivery module using the JNI interface. The Java Delivery module is configurable to allow data to be streamed into various targets. The supported targets for the Oracle GoldenGate Java Adapter product include JMS, file writing, and custom integrations. The Oracle GoldenGate for Big Data product includes all of those integrations and streaming capabilities to Big Data targets.

- Configuring a Replicat for Java Delivery (page 7-2)

7.2.1 Configuring a Replicat for Java Delivery

The Oracle GoldenGate Replicat process can be configured to send transaction data to the Oracle GoldenGate for Java module. Replicat consumes a local trail (for example `dirdat/aa`) and sends the data to the Java Delivery module. The Java module is responsible for processing all the data and applying it to the desired target.

Following is an example of adding a `Replicat` process:

```
ADD REPLICAT javarep, EXTTRAIL ./dirdat/aa
```

The process names and trail names used in the preceding example can be replaced with any valid name. Process names must be 8 characters or less, trail names must to be two characters. In the Replicat parameter file (`javarep.prm`), specify the location of the user exit library.

The Replicat process has transaction grouping built into the application. Transaction grouping can significantly improve performance when streaming data to a target database. Transaction grouping can also significantly improve performance when streaming data to Big Data applications. The Replicat parameter to control transaction grouping is the `GROUPTRANSOPS` variable in the Replicat configuration file. The default value of this variable is `1000` which means the Replicat process will attempt to group 1000 operations into single target transaction. Performance testing has generally shown that the higher the `GROUPTRANSOPS` the better the performance when streaming data to Big Data applications. Setting the `GROUPTRANSOPS` variable to `1` means that the original transaction boundaries from the source trail file (source database) will be maintained.

Table 7-1 User Exit Replicat Parameters

Parameter	Explanation
`REPLICAT javarep`	All Replicat parameter files start with the Replicat name
`SOURCEDEFS ./dirdef/tcust.def`	(Optional) If the input trail files do not contain the metadata records, the Replicat 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`).

Table 7-1 (Cont.) User Exit Replicat Parameters

Parameter	Explanation
`TARGETDB LIBFILE libggjava.so SET properties= dirprm/javarep.properties`	The `TARGETDB LIBFILE libggjava.so` parameter serves as a trigger to initialize the Java module. The `SET` clause to specify the Java properties file is optional. If specified, it should contain an absolute or relative path (relative to the Replicat executable) to the properties file for the Java module. The default value is *`replicat_name`*`.properties` in the `dirprm` directory.
`MAP schema.*, TARGET *.*;`	The tables to pass to the Java module; tables not included will be skipped. If mapping from source to target tables is required, one can use the *`MAP source_specification TARGET target_specification`* as describe in "Mapping and Manipulating Data" in *Administering Oracle GoldenGate*.
`GROUPTRANSOPS 1000`	Group source transactions into a single larger target transaction for improved performance. `GROUPTRANSOPS` of 1000 is the default setting. `GROUPTRANSOPS` sets a minimum value rather than an absolute value, to avoid splitting apart source transactions. Replicat waits until it receives all operations from the last source transaction in the group before applying the target transaction. For example, if transaction 1 contains 200 operations, and transaction 2 contains 400 operations, and transaction 3 contains 500 operations, then Replicat transaction contains all 1,100 operations even though `GROUPTRANSOPS` is set to the default of 1,000. Conversely, Replicat might apply a transaction before reaching the value set by `GROUPTRANSOPS` if there is no more data in the trail to process.

7.3 Configuring the Java Handlers

The Handlers are integrations with target applications which plug into the Oracle GoldenGate Java Delivery module. It is the Java Handlers which provide the functionality to push data to integration targets such as JMS or Big Data applications. The Java Adapter properties file is used to configure Java Delivery and Java handlers. To test the configuration, users may use the built-in file handler. 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=tx2xml.vm
gg.handler.myhandler.file=output.xml
```

This property file declares the following:

- Active event handlers. In the example 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. In the example `myhandler` is declared to be a `file` type of handler: `gg.handler.myhandler.type=file`

> **Note:**
>
> See the documentation for each type of handler (for example, the JMS handler or the file writer handler) for the list of valid properties that can be set.

- The format of the output is defined by the Velocity template `tx2xml.vm`. You may specify your own custom template to define the message format; just specify the path to your template relative to the Java classpath.

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 keywords: `jms_text` (or `jms`), `jms_map`, `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. Oracle GoldenGate Big Data package also contains built in Big Data target types. For more information, see *Integrating Oracle GoldenGate for Big Data*.

> **Note:**
>
> See the documentation for each type of handler (for example, the JMS handler or the file writer handler) for the list of valid properties that can be set.

8
Running Java Delivery

This chapter includes the following sections.

Topics:

- Starting the Application (page 8-1)
- Restarting the Java Delivery (page 8-1)

8.1 Starting the Application

To run the Java Delivery and execute the Java application, you only need an existing Oracle GoldenGate trail file. If the trail file does not contain metadata records, a source definitions file is also required to describe the schema for operations in the trail 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).

- Starting Using Replicat (page 8-1)

8.1.1 Starting Using Replicat

To run Java Delivery using Replicat, simply start the Replicat process from GGSCI:

```
GGSCI> START REPLICAT javarep
GGSCI> INFO REPLICAT javarep
```

The `INFO` command returns information similar to the following:

```
REPLICAT JAVAREP            Last Started 2015-09-10 17:25 Status RUNNING
Checkpoint Lag              00:00:00 (updated 00:00:00 ago)
Log Read Checkpoint File    ./dirdat/aa0000002015-09-10 17:50:41.000000
                             RBA 2702
```

8.2 Restarting the Java Delivery

There are two possible checkpoint files when running with Replicat, the Replicat process checkpoint file and the Java Delivery checkpoint file. Both files are located in the `dirchk` directory and created using the following naming conventions.

Replicat checkpoint file
group_name.cpr

Java delivery checkpoint file:
group_name.cpj

To suppress the creation and use of the Java Delivery checkpoint the Replicat process should be created using the following syntax:

```
ADD REPLICAT myrep EXTTRAIL ./dirdat/tr NODBCHECKPOINT
```

It is the `NODBCHECKPOINT` syntax that disables the creation and use of the Java Delivery checkpoint file.

- Restarting Java Delivery in Replicat (page 8-2)

8.2.1 Restarting Java Delivery in Replicat

The checkpoint handling in `Replicat` is more straightforward as it includes logic to pick which one out of the two checkpoint information is of higher priority. The logic is as follows:

- If the Java Delivery is started after user manually performed an `ADD` or `ALTER REPLICAT`, then the checkpoint information held by `Replicat` process will be used as the starting position.
- If the Java Delivery is started without prior manual intervention to alter checkpoint (for example, upon graceful stop or an abend), then the checkpoint information held by Java module will be used as the starting position.

  For example, restarting a Java Delivery using `Replicat` at the beginning of a trail looks like the following:

  1. Reset the `Replicat` to the beginning of the trail data:

     ```
     GGSCI> ALTER REPLICAT JAVAREP, EXTSEQNO 0, EXTRBA 0
     ```

  2. Reset the `Replicat`

     ```
     GGSCI> START JAVAREP
     GGSCI> INFO JAVAREP
     REPLICAT   JAVAREP    Last Started 2015-09-10 17:25   Status RUNNING
     Checkpoint Lag       00:00:00 (updated 00:00:00 ago)
     Log Read Checkpoint  File ./dirdat/aa000000
     2015-09-10 17:50:41.000000  RBA 2702
     ```

     It may take a few seconds for the Replicat 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 JAVAREP
     ```

     In the case where the Java Delivery is restarted after a crash or an abend, the last position kept by the Java module will be used when the application restarts.

9
Configuring Event Handlers

This chapter discusses types of event handlers explaining how to specify the event handler to use and what your options are. It explains how to format the output and what you can expect from the Oracle GoldenGate Report file.
This chapter includes the following sections.

Topics:

- Specifying Event Handlers (page 9-1)
- JMS Handler (page 9-2)
- File Handler (page 9-3)
- Custom Handlers (page 9-3)
- Formatting the Output (page 9-3)
- Reporting (page 9-4)

9.1 Specifying Event Handlers

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

- The Oracle GoldenGate Replicat or Extract process reads local trail data and passes the transactions, operations and database metadata to the Java Delivery Module. Metadata can come from the trail itself, a source definitions file.
- 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:

- Various built in Big Data handlers to apply records to supported Big Data targets, see Introduction to GoldenGate for Big Data to configure this type of handler.
- JMS message handlers to send to a JMS provider using either a `MapMessage`, or using a `TextMessage` with customized formatters.
- A specialized message handler to send JMS messages to Oracle Advanced Queuing (AQ).
- A file writer handler, for writing to a single file, or a rolling file.

> **Note:**
>
> The file writer handler is particularly useful as development utility, since the file writer can take the exact same formatter as the JMS `TextMessage` handler. Using the file writer 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.{name}.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 *name*. This *name* is used in the property file to define active handlers and set their properties; it is user-defined.

Implementation note (for Java developers): Following the preceding 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:

```
# 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.name.properties=foo.properties`.

9.2 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 types of JMS handlers can be specified:

- **jms** – 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_text` is a synonym for `jms`.)
- **aq** – sends text messages to Oracle Advanced Queuing (AQ). The `aq` handler is a `jms` handler configured for delivery to AQ. The messages can be formatted using Velocity templates or a custom formatter.
- **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

9.3 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=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 using the classpath.

9.4 Custom Handlers

For information on coding a custom handler, see Coding a Custom Handler in Java (page 11-4).

9.5 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 Filtering Events (page 11-1).

9.6 Reporting

Summary statistics about the throughput and amount of data processed are generated when the Replicat or Extract process stops. Additionally, statistics can be written periodically either 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 log files.

10
Java Delivery Properties

This chapter includes the following sections.

Topics:

- Common Properties (page 10-1)
- Delivery Properties (page 10-3)
- Java Application Properties (page 10-5)

10.1 Common Properties

The following properties are common to Java Delivery using either Replicat or Extract.

- Logging Properties (page 10-1)
- JVM Boot Options (page 10-2)

10.1.1 Logging Properties

Logging is controlled by the following properties.

- gg.log (page 10-1)
- gg.log.level (page 10-2)
- gg.log.file (page 10-2)
- gg.log.classpath (page 10-2)

10.1.1.1 gg.log

Specifies the type of logging that is to be used. The default implementation for the Oracle GoldenGate Adapters is the `jdk` option. This is the built-in Java logging called `java.util.logging` (`JUL`). The other logging options are `log4j` or `logback`.

For example, to set the type of logging to `log4j`:

```
gg.log=log4j
```

The recommended setting is `log4j`. The log file is created in the `dirrpt` subdirectory of the installation. The default log file name includes the group name of the associated `Extract` and the file extension is `.log`.

```
<process name>_<log level>_log4j.log
```

Therefore if the Oracle GoldenGate Replicat process is called `javaue`, and the `gg.log.level` is set to `debug`, the resulting log file name will be:

```
javaue_debug_log4j.log
```

10.1.1.2 gg.log.level

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

```
gg.log.level={ERROR|WARN|INFO|DEBUG|TRACE}
```

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.
- `TRACE` - Highest level of logging, includes all messages.

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
gg.log.level=INFO
```

10.1.1.3 gg.log.file

Specifies the path to the log file. The syntax is:

```
gg.log.file=path_to_file
```

Where the *`path_to_file`* is the fully defined location of the log file. This allows a change to the name of the log, but you must include the Replicat name if you have more than one Replicat to avoid one overwriting the log of the other.

10.1.1.4 gg.log.classpath

Specifies the classpath to the JARs used to implement logging. This configuration property is not typically used as the `ggjava.jar` library includes the required logging dependency libraries.

```
gg.log.classpath=path_to_jars
```

10.1.2 JVM Boot Options

The following options configure the Java Runtime Environment. Java classpath and memory options are configurable.

- jvm.bootoptions (page 10-2)

10.1.2.1 jvm.bootoptions

Specifies the initial Java classpath and other boot options that will be applied when the JVM starts. The `java.class.path` needs colon (:) separators for UNIX/Linux and semicolons (;) for Windows. This is where to specify various options for the JVM, such as initial and maximum 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. If using a separate `log4j` properties file then the location of the properties file must be included in the `bootoptions` `java.class.path` included in the bootoptions variable.
- **-verbose:jni**: run in verbose mode (for JNI)

For example (all on a single line):

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

The `log4j.configuration` property identifies a log4j properties file that is resolved by searching the classpath. You may use your own log4j configuration, or one of the preconfigured log4j settings: `log4j.properties` (default level of logging), `debug-log4j.properties` (debug logging) or `trace-log4j.properties` (very verbose logging). To use `log4j` logging with the `Replicat` process `gg.log=log4j` must be set.

Use of the one of the preconfigured log4j settings does not require any change to the classpath since those files are already included in the classpath. The `-Djava.class.path` variable must include the path to the directory containing a custom log4j configuration file without the * wild card appended.

10.2 Delivery Properties

The following properties are available to Java Delivery:

- General Properties (page 10-3)
- Statistics and Reporting (page 10-4)

10.2.1 General Properties

The following properties apply to all writer configurations:

- goldengate.userexit.writers (page 10-3)
- goldengate.userexit.chkptprefix (page 10-4)
- goldengate.userexit.nochkpt (page 10-4)
- goldengate.userexit.usetargetcols (page 10-4)

10.2.1.1 goldengate.userexit.writers

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

For example:

```
goldengate.userexit.writers=jvm
```

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

10.2.1.2 goldengate.userexit.chkptprefix

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

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

10.2.1.3 goldengate.userexit.nochkpt

Disables or enables the checkpoint file. The default is `false`, the checkpoint file is `enabled`. Set this property to `true` if transactions are supported and enabled on the target.

For example, Java Adapter Properties if JMS is the target and JMS local transactions are `enabled` (the default), set `goldengate.userexit.nochkpt=true` to disable the user exit checkpoint file. If JMS transactions are disabled by setting `localTx=false` on the handler, the checkpoint file should be enabled by setting `goldengate.userexit.nochkpt=false.`

```
goldengate.userexit.nochkpt=true|false
```

10.2.1.4 goldengate.userexit.usetargetcols

Specifies whether or not mapping to target columns is allowed. The default is `false`, no target mapping.

```
goldengate.userexit.usetargetcols=true|false
```

10.2.2 Statistics and Reporting

Disables or enables the checkpoint file handling. 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 Replicat module and those obtained from the Java module. The reports received from the Java side are formatted and returned by the individual handlers.

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

- jvm.stats.display (page 10-4)
- jvm.stats.full (page 10-5)
- jvm.stats.time | jvm.stats.numrecs (page 10-5)

10.2.2.1 *jvm*.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.

```
jvm.stats.display=true
```

10.2.2.2 *jvm*.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.

```
jvm.stats.full=true
```

10.2.2.3 *jvm*.stats.time | *jvm*.stats.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:

```
jvm.stats.time=600
jvm.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.

10.3 Java Application Properties

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

- Properties for All Handlers (page 10-5)
- Properties for Formatted Output (page 10-6)
- Properties for CSV and Fixed Format Output (page 10-8)
- File Writer Properties (page 10-12)
- JMS Handler Properties (page 10-12)
- JNDI Properties (page 10-16)
- General Properties (page 10-16)
- Java Delivery Transaction Grouping (page 10-17)

10.3.1 Properties for All Handlers

The following properties apply to all handlers:

- gg.handlerlist (page 10-6)
- gg.handler.name.type (page 10-6)

10.3.1.1 gg.handlerlist

The handler list is a list of active handlers separated by commas. 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`.

10.3.1.2 gg.handler.*name*.type

This type of handler. This 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|aq|singlefile|rollingfile|custom_java_class}
```

Where:

All but the last are pre-defined handlers:

- **jms** – Sends transactions, operations, and metadata as formatted messages to a JMS provider
- **aq** – Sends transactions, operations, and metadata as formatted messages to Oracle Advanced Queuing (AQ)
- **jms_map** – Sends JMS map messages
- **singlefile** – Writes to a single file on disk, but does not roll the file
- **rollingfile** – Writes transactions, operations, and metadata to a file on disk, rolling the file over after a certain size, amount of time, or both. For example:

  ```
  gg.handler.name1.rolloverSize=5000000
  gg.handler.name1.rolloverTime=1m
  ```

- *custom_java_class* – Any class that extends the Oracle GoldenGate for Java `AbstractHandler` class and can handle transaction, operation, or metadata events

  The Oracle GoldenGate foe Big Data package also contains more predefined handlers to write to various Big Data targets.

10.3.2 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 `aq` handler
- The `singlefile` and `rolling` handlers, for writing formatted output to files
- The predefined Big Data handlers
- gg.handler.name.format (page 10-7)
- gg.handler.name.includeTables (page 10-7)
- gg.handler.name.excludeTables (page 10-8)
- gg.handler.name.mode, gg.handler.name.format.mode (page 10-8)

10.3.2.1 gg.handler.*name*.format

Specifies the format used to transform operations and transactions into messages sent to JMS, to the Big Data target 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
  - `xml2` – internal XML format

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 installation directory is in the classpath, so the preceding 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.

10.3.2.2 gg.handler.*name*.includeTables

Specifies a list of tables this handler will include.

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 comma separated list of tables can be specified. For example, to have the handler only process tables `foo.customer` and `bar.orders`:

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

If the catalog and schema (or owner) of the table are specified, then only that catalog and schema matches the table name; otherwise, the table name matches any catalog and schema. A comma separated list of tables can be specified. For example, to have the handler only process tables `dbo.foo.customer` and `dbo.bar.orders`:

```
gg.handler.myhandler.includeTables=dbo.foo.customer, dbo.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.

10.3.2.3 gg.handler.*name*.excludeTables

Specifies a list of tables this handler will exclude.

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
```

If the catalog and schema (or owner) of the table are specified, then only that catalog and schema matches the table name; otherwise, the table name matches any catalog and 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 catalog `dbo` and schema `bar`:

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

10.3.2.4 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 `gg.handler.name.format.mode` when you have a custom formatter.

10.3.3 Properties for CSV and Fixed Format Output

If the handler is set to use either comma separated values (CSV) `CSV` or `fixed` format output, the following properties may also be set.

- gg.handler.name.format.delim (page 10-9)
- gg.handler.name.format.quote (page 10-9)
- gg.handler.name.format.metacols (page 10-9)
- gg.handler.name.format.missingColumnChar (page 10-10)
- gg.handler.name.format.presentColumnChar (page 10-10)
- gg.handler.name.format.nullColumnChar (page 10-10)

- gg.handler.name.format.beginTxChar (page 10-10)
- gg.handler.name.format.middleTxChar (page 10-10)
- gg.handler.name.format.endTxChar (page 10-10)
- gg.handler.name.format.wholeTxChar (page 10-11)
- gg.handler.name.format.insertChar (page 10-11)
- gg.handler.name.format.updateChar (page 10-11)
- gg.handler.name.format.deleteChar (page 10-11)
- gg.handler.name.format.truncateChar (page 10-11)
- gg.handler.name.format.endOfLine (page 10-11)
- gg.handler.name.format.justify (page 10-11)
- gg.handler.name.format.includeBefores (page 10-12)

10.3.3.1 gg.handler.*name*.format.delim

Specifies the delimiter to use between fields. Set this to no value to have no delimiter used. For example:

```
gg.handler.handler1.format.delim=,
```

10.3.3.2 gg.handler.*name*.format.quote

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

```
gg.handler.handler1.format.quote='
```

10.3.3.3 gg.handler.*name*.format.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, or delete records (see: `insertChar`, `updateChar`, `deleteChar`)
- **txind** – transaction indicator – such as `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
- **catalog** – catalog of the schema for the record
- **schema** – schema/owner of the table for the record
- **tableonly** – just table (no schema/owner)
- **table** – full name of table, `catalog.schema.table`
- **timestamp** – commit timestamp of record

For example:

```
gg.handler.handler1.format.metacols=opcode, table, txind, position
```

10.3.3.4 gg.handler.*name*.format.missingColumnChar

Specifies a special column prefix for a column value that was not captured from the source database transaction log. The column value is not in trail and it is unknown if it has a value or is `NULL`

The character used to represent the missing state of the column value can be customized. For example:

```
gg.handler.handler1.format.missingColumnChar=M
```

By default, the missing column value is set to an empty string and does not show.

10.3.3.5 gg.handler.*name*.format.presentColumnChar

Specifies a special column prefix for a column value that exists in the trail and is not `NULL`.

The character used to represent the state of the column can be customized. For example:

```
gg.handler.handler1.format.presentColumnChar=P
```

By default, the present column value is set to an empty string and does not show.

10.3.3.6 gg.handler.*name*.format.nullColumnChar

Specifies a special column prefix for a column value that exists in the trail and is set to `NULL`.

The character used to represent the state of the column can be customized. For example:

```
gg.handler.handler1.format.nullColumnChar=N
```

By default, the null column value is set to an empty string and does not show.

10.3.3.7 gg.handler.*name*.format.beginTxChar

Specifies the header metadata character (see `metacols`) used to identify a record as the `begin` of a transaction. For example:

```
gg.handler.handler1.format.beginTxChar=B
```

10.3.3.8 gg.handler.*name*.format.middleTxChar

Specifies the header metadata characters (see `metacols`) used to identify a record as the `middle` of a transaction. For example:

```
gg.handler.handler1.format.middleTxChar=M
```

10.3.3.9 gg.handler.*name*.format.endTxChar

Specifies the header metadata characters (see `metacols`) used to identify a record as the `end` of a transaction. For example:

```
gg.handler.handler1.format.endTxChar=E
```

10.3.3.10 gg.handler.*name*.format.wholeTxChar

Specifies the header metadata characters (see `metacols`) used to identify a record as a complete transaction; referred to as a `whole` transaction. For example:

```
gg.handler.handler1.format.wholeTxChar=W
```

10.3.3.11 gg.handler.*name*.format.insertChar

Specifies the character to identify an insert operation. The default `I`.

For example, to use `INS` instead of `I` for insert operations:

```
gg.handler.handler1.format.insertChar=INS
```

10.3.3.12 gg.handler.*name*.format.updateChar

Specifies the character to identify an update operation. The default is `U`.

For example, to use `UPD` instead of `U` for update operations:

```
gg.handler.handler1.format.updateChar=UPD
```

10.3.3.13 gg.handler.*name*.format.deleteChar

Specifies the character to identify a delete operation. The default is `D`.

For example, to use `DEL` instead of `D` for delete operations:

```
gg.handler.handler1.format.deleteChar=DEL
```

10.3.3.14 gg.handler.*name*.format.truncateChar

Specifies the character to identify a truncate operation. The default is `T`.

For example, to use `TRUNC` instead of `T` for truncate operations:

```
gg.handler.handler1.format.truncateChar=TRUNC
```

10.3.3.15 gg.handler.*name*.format.endOfLine

Specifies the end-of-line character as:

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

For example:

```
gg.handler.handler1.format.endOfLine=CR
```

10.3.3.16 gg.handler.*name*.format.justify

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

```
gg.handler.handler1.format.justify=left
```

10.3.3.17 gg.handler.*name*.format.includeBefores

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

```
gg.handler.handler1.format.includeBefores=false
```

10.3.4 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 (page 10-12)
- gg.handler.name.append (page 10-12)
- gg.handler.name.rolloverSize (page 10-12)

10.3.4.1 gg.handler.*name*.file

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

10.3.4.2 gg.handler.*name*.append

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

10.3.4.3 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=5MB
```

The default rollover size is `10MB`.

10.3.5 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=weblogic.properties
```

Just as with Velocity templates and formatting property files, this additional JMS properties file is found in the classpath. The preceding properties file `weblogic.properties` would be found in `{gg_install_dir}/dirprm/weblogic.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 (`weblogic.properties` in the preceding example). 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.handlerlist=myjms1,myjms2
gg.handler.myjms1.type=jms_text
gg.handler.myjms1.destination=queue.sampleA
gg.handler.myjms1.format=sample.vm
gg.handler.myjms1.properties=tibco-default.properties
gg.handler.myjms2.type=jms_map
gg.handler.myjms2.destination=queue.sampleB
gg.handler.myjms2.properties=tibco-default.properties
```

To set a property, specify the handler name as a prefix; for example:

```
gg.handlerlist=sample
gg.handler.sample.type=jms_text
gg.handler.sample.format=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.useJndi=false
gg.handler.sample.connectionFactory=ConnectionFactory
gg.handler.sample.connectionFactoryClass=\
    org.apache.activemq.ActiveMQConnectionFactory
gg.handler.sample.timeToLive=50000
```

- Standard JMS Settings (page 10-13)
- Group Transaction Properties (page 10-16)

10.3.5.1 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`).

- gg.handler.name.destination (page 10-14)
- gg.handler.name.user (page 10-14)
- gg.handler.name.password (page 10-14)
- gg.handler.name.queueOrTopic (page 10-14)
- gg.handler.name.persistent (page 10-14)
- gg.handler.name.priority (page 10-14)
- gg.handler.name.timeToLive (page 10-15)
- gg.handler.name.connectionFactory (page 10-15)
- gg.handler.name.useJndi (page 10-15)
- gg.handler.name.connectionUrl (page 10-15)
- gg.handler.name.connectionFactoryClass (page 10-15)
- gg.handler.name.localTX (page 10-15)

- gg.handlerlist.nop (page 10-15)
- gg.handler.name.physicalDestination (page 10-16)

10.3.5.1.1 gg.handler.*name*.destination

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.

```
gg.handler.name.destination=queue_or_topic
```

10.3.5.1.2 gg.handler.*name*.user

(Optional) User name required to send messages to the JMS server.

```
gg.handler.name.user=user_name
```

10.3.5.1.3 gg.handler.*name*.password

(Optional) Password required to send messages to the JMS server

```
gg.handler.name.password=password
```

10.3.5.1.4 gg.handler.*name*.queueOrTopic

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. This property is an alias of `gg.handler.name.destination`. The syntax is:

```
gg.handler.name.queueOrTopic=queue|topic
```

Where:

- `queue` – a message is removed from the queue once it has been read. This is the default.
- `topic` – messages are published and may be delivered to multiple subscribers.

10.3.5.1.5 gg.handler.*name*.persistent

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. The syntax is:

```
gg.handler.name.persistent={true|false}
```

10.3.5.1.6 gg.handler.*name*.priority

JMS defines a 10 level priority value, with 0 as the lowest and 9 as the highest. Priority is set to 4 by default. The syntax is:

```
gg.handler.name.priority=integer
```

For example:

```
gg.handler.name.priority=5
```

10.3.5.1.7 gg.handler.*name*.timeToLive

The length of time in milliseconds from its dispatch time that a produced message should be retained by the message system. A value of zero specifies the time is unlimited. The default is zero. The syntax is:

```
gg.handler.name.timeToLive=milliseconds
```

For example:

```
gg.handler.name.timeToLive= 36000
```

10.3.5.1.8 gg.handler.*name*.connectionFactory

Name of the connection factory to lookup using JNDI. `ConnectionFactoryJNDIName` is an alias. The syntax is:

```
gg.handler.name.connectionFactory=JNDI_name
```

10.3.5.1.9 gg.handler.*name*.useJndi

If `gg.handler.name.usejndi` is `false`, then JNDI is not used to configure the JMS client. Instead, factories and connections are explicitly constructed. The syntax is:

```
gg.handler.name.useJndi=true|false
```

10.3.5.1.10 gg.handler.*name*.connectionUrl

Connection URL is used only when not using JNDI to explicitly create the connection. The syntax is:

```
gg.handler.name.connectionUrl=url
```

10.3.5.1.11 gg.handler.*name*.connectionFactoryClass

The Connection Factory Class is used to access a factory only when not using JNDI. The value of this property is the Java class name to instantiate; constructing a factory object explicitly.

```
gg.handler.name.connectionFactoryClass=java_class_name
```

10.3.5.1.12 gg.handler.*name*.localTX

Specifies whether or not local transactions are used. The default is `true`, local transactions are used. The syntax is:

```
gg.handler.name.localTX=true|false
```

10.3.5.1.13 gg.handlerlist.nop

Disables the sending of JMS messages to allow testing of message generation. This is a global property used only for testing. The events are still generated and handled and the message is constructed. The default is `false`; do not disable message send. The syntax is:

```
gg.handlerlist.nop=true|false
```

Users can take advantage of this option to measure the performance of trail records processing without involving the handler module. This approach can narrow down the possible culprits of a suspected performance issue while applying trail records to the target system.

10.3.5.1.14 gg.handler.*name*.physicalDestination

Name of the queue or topic object, obtained through the `ConnectionFactory` API instead of the JNDI provider.

```
gg.handler.name.physicalDestination=queue_name
```

10.3.5.2 Group Transaction Properties

These properties set limits for grouping transactions.

10.3.6 JNDI Properties

These JNDI properties are required for connection to an Initial Context to look up the connection factory and initial destination.

```
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
```

10.3.7 General Properties

The following are general properties that are used for the Java framework:

- gg.classpath (page 10-16)
- gg.report.time (page 10-17)
- gg.binaryencoding (page 10-17)

10.3.7.1 gg.classpath

Specifies a comma delimited list of additional paths to directories or JARs to add to the classpath. Optionally, the list can be delimited by semicolons for Windows systems or by colons for UNIX. For example:

```
gg.classpath=C:\Program Files\MyProgram\bin;C:\Program Files\ProgramB\app\bin;
```

This Adapter properties file configuration property should be used to configure pathing to custom Java JARs or to the external dependencies of Big Data applications.

10.3.7.2 gg.report.time

Specifies how often statistics are calculated and sent to Extract for the processing report. If Extract is configured to print a report, these statistics are included. The syntax is:

```
gg.report.time=report_interval{s|m|h}
```

Where:

- *report_interval* is an integer
- The valid time units are:
  - `s` - seconds
  - `m` - minutes
  - `h` - hours

If no value is entered, the default is to calculate and send every 24 hours.

10.3.7.3 gg.binaryencoding

Specifies the binary encoding type. The desired output encoding for binary data can be configured using this property. For example:

```
gg.binaryencoding=base64|hex
```

The default value is base64. The valid values to represent binary data are:

- `base64` - a base64 string
- `hex` - a hexadecimal string

10.3.8 Java Delivery Transaction Grouping

Transaction grouping can significantly improve the performance of Java integrations especially Big Data integrations. Java Delivery provides functionality to perform transaction grouping. When Java Delivery is hosted by a Replicat process then the `GROUPTRANSOPS` Replicat configuration should be used to perform transaction grouping.

11

Developing Custom Filters, Formatters, and Handlers

This chapter includes the following sections.

Topics:

- Filtering Events (page 11-1)
- Custom Formatting (page 11-1)
- Coding a Custom Handler in Java (page 11-4)
- Additional Resources (page 11-6)

11.1 Filtering Events

By default, all transactions, operations and metadata events are passed to the `DataSourceListener` event handlers. An event filter can be implemented to filter the events sent to the handlers. The filter could select certain operations on certain tables containing certain column values, for example

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=mytemplate.vm
gg.handler.one.filter=com.mycompany.MyFilter
```

To activate the filter, you write the filter and set it on the handler; no additional logic needs to be added to specific handlers.

11.2 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 (page 11-1)
- Using a Velocity Template (page 11-3)

11.2.1 Coding a Custom Formatter in Java

The preceding examples show a JMS handler and a file output handler using the same formatter (`com.mycompany.MyFormatter`). The following is an example of how this formatter may be implemented.

Example 11-1 Custom Formatting Implementation

```
package com.mycompany.MyFormatter;
import oracle.goldengate.datasource.DsOperation;
import oracle.goldengate.datasource.DsTransaction;
import oracle.goldengate.datasource.format.DsFormatterAdapter;
import oracle.goldengate.datasource.meta.ColumnMetaData;
import oracle.goldengate.datasource.meta.DsMetaData;
import oracle.goldengate.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 g*g_install_dir*/dirprm:

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

The resulting class files are 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 using the user exit properties (using `java.class.path` in the `jvm.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/*
```

11.2.2 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
```

Update the Java application properties file to use the template:

```
# set properties on 'one'
gg.handler.one.type=file
gg.handler.one.format=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
```

11.3 Coding a Custom Handler in Java

A custom handler can be implemented by extending `AbstractHandler` as in the following example:

```
import oracle.goldengate.datasource.*;
import static oracle.goldengate.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..."; }
        @Override
        public Status ddlOperation(OpType opType, ObjectType objectType, String
objectName, String ddlText) }
```

The method in `AbstractHandler` is not abstract rather it has a body. In the body it performs cached metadata invalidation by marking the metadata object as dirty. It also provides TRACE-level logging of DDL events when the `ddlOperation` method is specified. You can override this method in your custom handler implementations. You should always call the super method before any custom handling to ensure the functionality in `AbstractHandler` is executed

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 then received. If the Java module 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 (this is 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 that contains all processed table metadata. 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 when Extract is processing with compressed updates.) The column values may contain both before and after values.

   For the `ddlOperation` method, the options are:

   - `opType` - Is an enumeration that identifies the DDL operation type that is occurring (`CREATE`, `ALTER`, and so on).
   - `objectType` - Is an enumeration that identifies the type of the target of the DDL (`TABLE`, `VIEW`, and so on).
   - `objectName` - Is the fully qualified source object name; typically a fully qualified table name.
   - `ddlText` - Is the raw DDL text executed on the source relational database.
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 (the number of operations andtransactions processed and other details).

An example of a simple printer handler, which just prints out very basic event information for transactions, operations and metadata follows. The handler also has a property `myoutput` for setting the output file name; 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 files in the JVM classpath using the Java properties (using `java.class.path` in the `jvm.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 extracted 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 in the preceding example.

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 preceding 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. For custom types, you may create a custom property editor to convert the String to your custom type.

11.4 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 customizing and extension.

In each package, some classes have been intentionally omitted for clarity. The important classes are:

- `oracle.goldengate.datasource.DsTransaction`: represents a database transaction. A transaction contains zero or more operations.
- `oracle.goldengate.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.
- `oracle.goldengate.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).
  - `oracle.goldengate.datasource.DsColumnComposite` is the composite
  - `oracle.goldengate.datasource.DsColumnBeforeValue` is the column value before the operation (this is optional, and may not be included in the operation)
  - `oracle.goldengate.datasource.DsColumnAfterValue` is the value after the operation
- `oracle.goldengate.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.
- `oracle.goldengate.datasource.meta.TableMetaData`: represents all metadata for a single table; contains zero or more `ColumnMetaData`.
- `oracle.goldengate.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.

Part IV

Troubleshooting the Oracle GoldenGate Adapters

This part of the book provides information on troubleshooting problems with the Oracle GoldenGate Adapters.

Topics:

- Troubleshooting the Java Adapters (page 12-1)

12
Troubleshooting the Java Adapters

This chapter includes the following sections:

Topics:

- Checking for Errors (page 12-1)
- Reporting Issues (page 12-2)

12.1 Checking for Errors

There are two types of errors that can occur in the operation of Oracle GoldenGate for Java:

- The Replicat process running or VAM does not start or abends
- The process runs successfully, but the data is incorrect or nonexistent

If the Replicat or 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 {replicat/extract name}
   ```

2. Check the applicable log file.

   For the native log file:

   - Look at the last messages reported in the log file for the native 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 the only log file for the shared library, not the Java application.

3. If the Replicat, 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:

  ```
  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 native 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> REPLICAT 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 Replicat from the beginning of a trail, see Restarting the Java Delivery (page 8-1).

12.2 Reporting Issues

If you have a support account for Oracle GoldenGate, submit a support ticket and include the following:

- Operating system and Java versions

  The version of the Java Runtime Environment can be displayed by:

  ```
  $ java -version
  ```

- Configuration files:
  - Parameter file for the Replicat
  - All properties files used, including any JMS or JNDI properties files
  - Velocity templates for formatting purposes
  - If applicable, also include the target-specific configuration file
- Log files:

  In the Oracle GoldenGate install directory, all `.log` files: the Java `log4j` log files and the native module or VAM log file.

A
List of Included Examples

The following examples are located in the `AdaptersExamples` subdirectory of the installation location.

Flat File Writer

- Using the Oracle GoldenGate Flat File Adapter to convert Oracle GoldenGate trail data to text files.

Message Delivery

- Using the Oracle GoldenGate Java Adapter to send JMS messages with a custom message format.
- Using the Oracle GoldenGate Java Adapter to send JMS messages with custom message header properties.

Message Capture

- Using the Oracle GoldenGate Java Adapter to process JMS messages, creating an Oracle GoldenGate trail.

Java API

- Using the Oracle GoldenGate Java Adapter API to write a custom event handler.

B
Customizing Logging

This example describes how to customize the logging for 11.2.1 and greater Oracle GoldenGate adapters by using one of two methods:

- Use Java adapter properties

  ```
  gg.log={ jdk | logback | log4j }
  gg.log.level={ info | debug | trace }
  gg.log.classpath={ classpath for logging }
  ```

  If the log implementation property `gg.log` is not set, the `jdk` option defaults. This specifies that `java.util.logging` (`JUL`) is used. The log level defaults to `info`. To customize this, you can set the `gg.log` to either:

  – `log4j` - This automatically configures the classpath to include the Log4j and appropriate `slf4j-log4j` binding.

  – `logback` - To use the `logback` option, the logback JARS must be manually downloaded and copied into the install directory. The classpath is still automatically configured as long as the JARs are copied into the predefined location. See `ggjava/resources/lib/optional/logback/ReadMe-logback.txt` for more information.

- Use JVM options

  Instead of using default logging or setting logging properties, `jvm.bootoptions` can be used to define the logging. To do this, set `jvm.bootoptions` to include the system property that defines the configuration file by doing one of the following:

  – Specify a `log4j` configuration file:

  ```
  jvm.bootoptions=-Dlog4j.configuration=my-log4j.properties
  ```

  This implicitly sets `gg.log` to `log4j` as the type of logging implementation and appends `slf4j-log4j12` binding to the classpath.

  – Specify a `java.util.logging` properties file or class:

  ```
  jvm.bootoptions=-Djava.util.logging.config.file=my-logging.properties
  ```

  This implicitly sets `gg.log=jdk`, which specifics `java.util.logging` (`JUL`). It appends `slf4j-jdk14` binding to the classpath.

  – First, download and copy `logback-core-jar` and `logback-classic-jar` into `ggjava/resources/lib/optional/logback`. Then specify a logback configuration file:

  ```
  jvm.bootoptions=-Dlogback.configuationFile=my-logback.xml
  ```

  This implicitly sets `gg.log=logback` and appends `logback-classic` and `logback-core` to the classpath.

  These are implicit settings of `gg.log` and `gg.log.classpath` that will be overridden by an explicit setting of either of these properties in the property file. The logging

classpath will also be overridden by setting the JVM classpath to include specific JARs, such as:

```
jvm.bootoptions=...-Djava.class.path=mypath/my1.jar:mypath2/my2.jar...
```

> **Note:**
>
> Setting the JVM classpath to include specific JARs may cause duplicate, possibly conflicting, implementations in the classpath.

Index

A

ActiveMQ, *9-2*
application
    start, *8-1*
aq, *9-2*

C

column data, *5-1*, *5-3*, *5-4*
    fixed width parsing, *5-4*
comma-separated values, *10-8*
comments
    to identify key columns, *5-7*
    to specify date format, *5-6*
configure
    event handlers, *9-1*
connection factory, *10-15*
copybook
    definition for fixed width parsing, *5-5*
    for source and target definitions, *5-2*
CSV format, *10-8*

D

data definitions
    how to specify, *5-2*
delimited message
    basis for parsing, *5-9*
    format, *5-9*
    metadata columns, *5-9*
    parsing properties, *6-11*
    parsing rules, *5-10*

E

event handlers
    configuring, *9-1*
Extract
    adding the VAM Extract, *4-1*
    parameters for the VAM, *4-2*

F

file handler, *9-3*
file writer
    properties, *10-12*
fixed width message
    basis for parsing, *5-5*
    defining the header, *5-5*
    header and record data translation, *5-7*
    key identification, *5-7*
    parsing properties, *6-6*
    table name, *5-6*
    timestamp formats, *5-6*
fixed-format, *10-8*
formatting, *9-3*, *10-6*

G

Gendef utility, *5-2*, *5-19*

H

handler
    file, *9-3*
    JMS, *9-2*
    properties, *10-5*
handlers
    configuring, *9-1*
header
    defining for fixed width message, *5-5*

I

issues
    reporting, *12-2*

J

Java application
    properties, *10-5*
Java code, *11-3*
JBoss, *9-2*
jms, *9-2*
JMS

JMS *(continued)*
connecting to, *4-2*
properties, *6-2*
JMS handler, *9-2*
properties, *10-12*
JMS handler types
aq, *9-2*
jms, *9-2*
JMS messages
retrieving, *4-3*
JMS provider, *9-2*
JMS queue or topic, *10-14*
jms_map, *9-2*
JNDI, *4-2*, *6-4*
properties, *6-5*, *10-16*

K

key identification
for fixth width messages, *5-7*

L

logging properties, *6-1*, *10-1*

M

metadata columns
delimited message, *5-9*

O

operation type, *5-1*, *5-3*, *5-12*
for XML parsing, *5-12*
mapping, *5-6*
optype
specifying for fixed width parsing, *5-6*
Oracle GoldenGate
documentation, *1-5*
output, *9-3*, *10-6*

P

parameters
VAM Extract, *4-2*
parser
required data, *5-2*
role of, *5-1*
types, *5-1*
properties, *6-1*
delimited message parsing, *6-11*
file writer, *10-12*
fixed width message parsing, *6-6*
handlers, *10-5*
properties *(continued)*
Java application, *10-5*
JMS handler, *10-12*
JNDI, *6-5*, *10-16*
logging, *6-1*, *10-1*
User Exit, *6-1*, *10-1*
XML message parsing, *6-21*

Q

queue, *10-14*

R

reporting
issues, *12-2*

S

sequence identifier, *5-1*, *5-2*
Solace, *9-2*
source definitions file, *6-7*, *6-12*, *6-22*
generating, *5-2*, *5-19*
sourcedefs
type of fixed schema, *6-6*
start
application, *8-1*

T

table name, *5-1*, *5-3*, *5-9*, *5-11*, *5-16*
defining for fixed width message, *5-6*
for delimited parsing, *5-10*
for XML parsing, *5-12*
TIBCO, *9-2*
timestamp, *5-1*, *5-3*, *5-5*, *5-6*, *5-9*, *5-12*, *5-15*, *5-16*
formats for fixed width message, *5-6*
topic, *10-14*
TRANLOGOPTIONS
GETMETADATAFROMVAM option, *4-2*
VAMCOMPATIBILITY option, *4-2*
transaction
specifying boundary for, *5-4*, *5-15*
transaction identifier, *5-1*, *5-2*
for XML parsing, *5-15*
transaction indicator, *5-1*, *5-4*, *5-9*, *5-16*
transaction name, *5-1*, *5-4*
transaction owner, *5-1*, *5-4*

U

User Exit
properties, *6-1*, *10-1*

V

VAM parameter, *4-2*
Velocity template, *11-3*

W

WebLogic, *9-2*

X

XML message
XML message *(continued)*
basis for parsing, *5-11*
column rules, *5-17*
formatted in dynamic XML, *5-11*
formatted in static XML, *5-11*
operation rules, *5-16*
parsing properties, *6-21*
parsing rules, *5-12*
supported XPath expressions, *5-13*
transaction rules, *5-15*