Database Logger Programmers Guide[1]

Author: Jeff Gordy

[1]V 0.1

Contents

List of Tables

Chapter 1

Theory of Operation

jPOS is a wonderful tool to send, receive, parse, process, and track iso8583 messages. The primary method of data tracking is the jPOS logging subsystem that prints activity in an easily parsed XML format suitable inclusion in a log file, printing to standard out, or redirection back into jPOS for a quasi "instant replay".

However, many developers working with their applications need to track message information in a relational database for easy extraction at a later date. The jPOS Database Logger module enables a developer to quickly set up database tracking of iso8583 messages. The built–in tracking will probably be sufficient for most users, however it is very easy to extend the module to include additional tracking if desired.

The goal of this module is to provide an easy and fast way track iso8583 messages in a relational database without having to know how Java handles database interaction, or how jPOS handles iso8583 messages.

Chapter 2

Introduction

2.1 About This Document

For most experienced jPOS developers a quick look at the sample application chapter will be enough to see what this is all about.

For the rest of us it is beneficial to read the whole doc.

2.1.1 Style and Syntax

My writing style is simple. I write like I am talking to the reader. It is pleasant to have a one–way conversation because I do not get interrupted. Oh, you will ask questions, but I will anticipate them and answer them directly.

```
Code samples will be in this font.
```

Italics will be used for a variable within text, or in a command within text.

2.2 Introduction To jPOS Relational Database Logging

So you've decided this thing might be for you, but what does it do? Basically the code for this module is intended to sit in–between your

higher level business logic application and the particular interchange you are working with. This is the point where a jPOS "channel" is used to communicate with the interchange. Although this is a good place to grab messages for logging, you can perform the logging anywhere the messages are in scope.

At the first instantiation of DatabaseLogger a connection pool is created. Whenever a DatabaseLogger object needs a connection to the database it is leased temporarily from the pool. When finished it is given back. The connection pool greatly speeds up complex database applications. While this logger does not necessarily qualify as complex, it can't hurt.

In your business logic application you create a new DatabaseLogger object, and pass it ISOMsg objects that you wish to record. It is currently possible to track four separate complete messages for any given transaction, and an unlimited number of bits for any message you may be working with.

If you only want to record the messages, and do not care to separate out particular bits that is fine. Additionally it is possible to track bits only and not the complete message.

Chapter 3

Getting Started

Before you can run any of the code examples you have to get a few building blocks in place. First, you need a database server. If you do not currently have one I recommend the open source MySQL available at http://www.mysql.com. It's fast, and runs on almost any operating system.

After getting a working server, you need to build the jpos-dblogger.jar module. At the root tree of this project type *bin/build dblog*. Please note that you must have a working jpos jar file in your class path. The *dist* directory will contain the compiled jpos-dblogger.jar file. You then must put this somewhere in your class path, or in the *lib* directory of q2.

At this point you need to create the tables the Database Logger will work with. The *etc* directory contains two files **jposMessageDetail.sql** and **jposMessageLog.sql**. These two files contain CREATE statements for the tables. Before sending these commands directly to your server make sure the column types are supported. If they are not, you will need to change them to something that will work for your server. For example the **text** type fields would need to be changed to **long** if using Oracle 9i.

If using MySQL you can pass the sql statements directly to the server with the following command:

```
mysql -h hostname -u user -ppassword -t database <
    jposMessageDetail.log > errors.txt
```

With most other database servers you can copy and paste the command into the "query" screen.

After the tables are created you are ready to run the test examples or just start using the Database Logger in your code. Oh wait, one more step, you need to set up a configuration file with your JDBC connection information.

Chapter 4

JDBC Connection and Configuration File

4.1 What is JDBC?

JDBC (Java Database Connectivity) is a Java API for accessing tabular data. It consists of a set of classes and interfaces that provide standardized access to a database. It makes it easy to send Structured Query Language statement to the database server, and read the response if any. It is also possible to use the JDBC API to access flat files containing tabular data.

The JDBC API provides a bunch of standard methods that the various database server companies or groups have to implement. Actually, they don't *h*ave to implement all of them, but if they do their interface into their database is said to be "JDBC Compliant". Compliance can be checked by calling a standard method *jdbcCompliant()* and evaluating the boolean response.

This interface into the database is called a "JDBC Driver".

4.2 JDBC Driver

To make a connection to your database server you need a JDBC "driver" written for that server. If you don't have a driver for your server you can go here: http://servlet.java.sun.com/products/jdbc/drivers. With 209

drivers available at time of writing, I am sure you will find something that will work.

Each driver will come with its own installation instructions. If you are in a pinch, and the documentation is weak, you usually put the driver's jar file in your class path and you are ready to go.

4.3 Establishing a Connection

To make a connection to a particular database you will need to identify the driver used, a location and database name for your server (also known as the URL), your username, and your password.

The only funky part is the URL. JDBC defines the URL to look much like an Internet URL, which can be confusing and lame if you are used to other programmatic methods of connecting to a database. But, we have to learn it so here it goes:

4.3.1 The URL

The JDBC URL is made up of the keyword "jdbc" followed by a "subprotocol" followed by a "subname", all separated by colons.

The subprotocol is the name of the driver or the name of a databases connectivity mechanism, which may be supported by multiple drivers. The best multiple driver example is the subprotocol *odbc*. There are many odbc drivers out there, but they all use the subprotocol odbc. Whatever driver you are using should have the subprotocol specified in the documentation. In my case, I am using MySQL, and the driver defines the subprotocol as "mysql" so my connection URL through this point would be **jdbc:mysql:**. With that period omitted of course.

The subname is a way to identify the data source. The subname can vary widely based on the subprotocol and it's desired format. The point of the subname is to provide enough information so the JDBC driver manager can find the database. In my case, I am using a server running on my local machine, and am using the database "test". Thus my URL connection string is **jdbc:mysql://localhost/test**

Of course, it can get much more complicated than that. The MySQL driver allows me to specify the port number that my server is listening on,

as well as a few connection parameters. Adding that I get the following URL **jdbc:mysql://localhost:3306/test?autoReconnect=true**

Your mileage will vary.

4.4 Database Logger Configuration File

To enable you to change databases easily without re-compiling the Database Logging module, I have decided to use a configuration file scheme that is used quite a bit throughout the jPOS project. Basically is is a list value pair file that contains the necessary connection information described above.

For the examples to work you have to take the **test.cfg** file in the *etc* directory and edit it to reflect your specific connection information. In my case the text of my configuration file is as follows:

```
jdbc.driver=org.gjt.mm.mysql.Driver
jdbc.url=jdbc:mysql://localhost:3306/test?autoReconnect=true
jdbc.user=jgordy
jdbc.password=mypassword
```

Chapter 5

Database Table Structure

Ok, now that we have a working database installed, and a working connection configuration file, and our two tables created, we can actually start to use this stuff. But first let's go over the table structure to elucidate the design of the logger:

The jposMessageLog table contains four columns in which to store iso8583 messages. Each sequence of messages is assigned a message ID *msgId*. When your application calls the method *logIncomingMsg(ISOMsg)* the logger will get the next available message id from the database, update it's timestamp, and insert the message into the "incomingMsg" field. After that, all other messages are optional. If your application performs transformations on a message it is logical to record it in the "transformMsg" column. If it does not, you could go directly to your

Table 5.1: jposMessageLog Structure

jposMessageLog Structure			
Field	Type	Null	Key
msgId	bigint	NO	PRI
incomingMsg	text	YES	None
transformMsg	text	YES	None
replyMsg	text	YES	None
outgoingMsg	text	YES	None
dateAdd	datetime	YES	None

Table 5.2: jposMessageDetail Structure

jposMessageDetail Structure			
Field	Type	Null	Key
detailId	bigint	NO	PRI
msgId	bigint	YES	None
bit	varchar(3)	YES	None
value	varchar(255)	YES	None
msgType	ENUM	YES	None

interchange, grab your reply and store it. If you then send a copy of the reply or an edited reply back to your client application (if there is one), you can store it in the "outgoingMsg" field. In any event, each of the calls to store a message in one of the columns is not required.

What IS currently required is to maintain the current database table structure. I am working on a way to use metadata and an additional configuration file to determine what should be recorded, and in what structure, but that is yet to be implemented.

The jposMessageDetail table is quite simple. Each time a new detail bit is tracked a detailId is created for it. Then the msgId from the jposMessageLog table is inserted. Next we have the bit value from the array of bit strings passed to the constructor, and the value of that bit in the message. I know you have not seen the constructor yet, but we will get to it soon. The msgType is an enumerated field that matches the column names from the jposMessageLog table. The reason for this is to simplify research queries for reporting, etc. For example, having the msgType field enables queries like the following:

```
SELECT count(*) FROM jposMessageDetail where bit = '39'
and value = '00' AND msgType = 'outgoingMsg'
```

This query will find a count of all approvals that were delivered to a client application or terminal. With many terminals, a simple join on the msgId and specifying one of the bits used for tracking individual terminals will allow you to drill–down to the individual terminal level. For example, if we had two terminals that stored their terminal id in bit 41, and we tracked it

in the jposMessageDetail table we could count the approvals for terminal number 1 as follows:

```
SELECT count(*) FROM jposMessageDetail j1, jposMessageDetail j2
WHERE j1.msgID = j2.msgID
AND j1.bit = '41' AND j1.value = '1'
AND j2.bit = '39' AND j2.value = '00'
AND j1.msgType = 'outgoingMsg' AND j2.msgType = 'outgoingMsg'
```

The SQL for your server may be a bit different, but you get the general idea. Also, since this is not a SQL tutorial I'll stop with the queries.

Chapter 6

Internals

6.1 Data Access Objects

From Sun's Data Access Object Explanation Site:

> **Problem**
>
> Many real-world Java 2 Platform, Enterprise Edition (J2EE) applications need to use persistent data at some point. For many applications, persistent storage is implemented with different mechanisms, and there are marked differences in the APIs used to access these different persistent storage mechanisms. Other applications may need to access data that resides on separate systems. For example, the data may reside in mainframe systems, Lightweight Directory Access Protocol (LDAP) repositories, and so forth. Another example is where data is provided by services through external systems such as business-to-business (B2B) integration systems, credit card bureau service, and so forth.
>
> Typically, applications use shared distributed components such as entity beans to represent persistent data. An application is considered to employ bean-managed persistence (BMP) for its entity beans when these entity beans explicitly access the persistent storage-the entity bean includes code to directly access the persistent storage. An application with simpler requirements may forego using entity beans and instead use

session beans or servlets to directly access the persistent storage to retrieve and modify the data. Or, the application could use entity beans with container-managed persistence, and thus let the container handle the transaction and persistent details.

Applications can use the JDBC API to access data residing in a relational database management system (RDBMS). The JDBC API enables standard access and manipulation of data in persistent storage, such as a relational database. The JDBC API enables J2EE applications to use SQL statements, which are the standard means for accessing RDBMS tables. However, even within an RDBMS environment, the actual syntax and format of the SQL statements may vary depending on the particular database product. . . .

Solution

Use a Data Access Object (DAO) to abstract and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data.

The DAO implements the access mechanism required to work with the data source. The data source could be a persistent store like an RDBMS, an external service like a B2B exchange, a repository like an LDAP database, or a business service accessed via CORBA Internet Inter-ORB Protocol (IIOP) or low-level sockets. The business component that relies on the DAO uses the simpler interface exposed by the DAO for its clients. The DAO completely hides the data source implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components. Essentially, the DAO acts as an adapter between the component and the data source.

6.2 Database Logger DAO Use

The Database Logger module uses the Data Access Object pattern to implement the underlying database methods. The java code for the

database was generated by DaoGen from http://titaniclinux.net/daogen. The utility of using this product lies in end user database customization.

If you wish you may generate your own table structure using DaoGen, and you only have to modify or extend the DatabaseLogger object to get access to your new structure.

Chapter 7

A Sample Application

7.1 Fundamentals

Ok, now we are actually ready to see some code for this thing. The first call we have to make when starting our application is the creation of a new DatabaseLogger object. There are two forms. The first method is used when we only want to track complete messages, the second adds an array of strings representing bit numbers to track in the detail table.

```
DatabaseLogger db = new DatabaseLogger(cfg);
            or
String[] bits = {"11", "39"};
DatabaseLogger db = new DatabaseLogger(cfg, bits);
```

As stated earlier, this will take the configuration file, parse it, and open a connection pool to the database.

Once we have a ISOMsg message *m* that we wish to log we simply call:

```
db.logIncomingMessage(m);
```

This will create the entry in the jposMessageLog table, and if the bit constructor was used will log the appropriate bits for this message. If, for example, you are tracking messages and looking at the response code bit

39, you may note that your original or incoming message does not have a bit 39. Don't worry about the detail tracking. If the message does not have a bit no errors are thrown, it just skips that detail tracking entry.

Once we have a reply to our message we can track it with the following call:

```
db.logReplyMessage(reply);
```

It is just that simple. You call the method for whatever column you want to use. When finished, you can let the garbage collector take over, or explicitly close all open pool connections with:

```
db.freePool();
```

7.2 Complete Application

The following application is the text from the Test.java file in the examples directory.

```
package org.jpos.dblog.examples;

/**
 * <p>Title: jPOS Modules Example Script</p>
 *
 * <p>Description: JPOS Extensible Database Module</p>
 *
 * <p>Copyright: See terms of license at http://jpos.org/license.html</p>
 * @author Jeff Gordy
 * @version 1.0
 */

import org.jpos.iso.*;
import org.jpos.core.*;
import java.util.Date;
import java.util.Random;
import org.jpos.dblog.*;
```

```
public class Test {

  /**
   * interchange - In this we are pretending we have sent the message to an
   * outside interchange and are reading the reply off the wire.  All we do is
   * set the response code bit 39 to approved.
   *
   * @param m ISOMsg
   * @throws ISOException
   * @return ISOMsg
   */
  public static ISOMsg interchange(ISOMsg m) throws ISOException {
    ISOMsg c = (ISOMsg) m.clone();
    c.setResponseMTI();
    c.set(39, "00");
    return c;
  }

  /**
   * main - Here we create an ISOMsg object, send it to our dummy interchange,
   * and log both the original message and the result from the interchange.
   *
   * @param args String[]
   */
  public static void main(String[] args) {
    // First we want to assign our configuration file.
    Configuration cfg = null;
    try {
      cfg = new SimpleConfiguration("/home/jgordy/test.cfg");
    } catch (Exception e) {
      System.out.println("Error with Configuration File: " + e.getMessage());
    }

    try {
      Date d = new Date();
      // This random number will be used to show unique
      // numbers over mutliple runs in the tables.
      Random r = new Random(System.currentTimeMillis());
```

```
      int somethingToTrack = r.nextInt(10000) + 1;

      // create a new simple 800 messsage
      ISOMsg m = new ISOMsg("0800");
      m.set(11, "000001");
      m.set(12, ISODate.getTime(d));
      m.set(13, ISODate.getDate(d));
      m.set(41, Integer.toString(somethingToTrack) );

      // the bit string array indicates we want to track bits 11, 12, 39, and 41
      // if we did not create a bit string the database logger object would
      // just log complete messages.
      String[] bits = {"11", "12", "39", "41"};
      // create the new databaselogger object.  Pass it our configuration file
      // for the JDBC connection, and our bit array.
      DatabaseLogger db = new DatabaseLogger(cfg, bits);

      // log the message "m" as the incoming message.  The incoming message
      // creates a new entry in the jposMessageLog table, and gets assigned a
      // unique message id.
      db.logIncomingMessage(m);

      // here we read an ISOMsg m2 pretending it was a response from an interchange.
      ISOMsg m2 = interchange(m);
      // and we log the response as our outgoingMessage
      db.logOutgoingMessage(m2);

      // here we tell the DatabaseLogger object that we are done with it and it
      // can close all connections in the connection pool.  This should only really
      // be done whenever you shutdown your application.  Otherwise the connection po
      // needs to remain open to actually "pool"
      db.freePool();

    } catch (ISOException e) {
      e.printStackTrace();
    } catch (java.sql.SQLException e) {
      new SQLExceptionHandler(e);
    } catch (NotFoundException e) {
      e.printStackTrace();
    }
```

```
  }
}
```

Chapter 8

Integration with q2

To be written as I find a better way to integrate with q2 than a custom connector.

However, if you want to look at what is there now, you can use the DatabaseConnector.java file in the examples directory in your server.xml q2 file. It is a rip–off of the Connector.java file written by Alejandro, but performs some transformations and logs the messages to the database.

My 30_server.xml file looks like this:

```
<server class="org.jpos.q2.iso.QServer" logger="Q2" name="server"
    realm="xmlserver">
    <attr name="port" type="java.lang.Integer">8080</attr>

    <!-- ISOChannel listening server configuration -->
    <channel class="org.jpos.iso.channel.XMLChannel" logger="Q2"
        packager="org.jpos.iso.packager.XMLPackager">
    </channel>

    <request-listener class="org.jpos.dblog.examples.DatabaseConnector"
        logger="Q2">
        <property name="destination-mux" value="mux.my-mux" />
        <property name="timeout" value="60000" />
    </request-listener>
</server>
```

Chapter 9

Version Information

9.1 Revision History

version 0.1 This is the initial version of the Database Logging Guide.

9.2 References

REF 1 "JDBC API Tutorial and Reference, Third Edition" by Fisher, Ellis, and Bruce

REF 2 DaoGen http://titaniclinux.net/daogen

REF 3 Advanced Programming for the Java 2 Platform http://java.sun.com/developer/onlineTraining/Programming/JDCBook/index.html#c

REF 4 Data Access Objects http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

9.3 Copyright Notices