Chapter 13
JDBC And NOI

We've seen in the earlier chapters of this book how to use the Notes Object Interface (NOI) from Java programs (applications, Servlets, and Agents) to manipulate Domino objects to our advantage. In this chapter, we'll look at how to integrate the use of NOI with other Java APIs, specifically the one from JavaSoft known as JDBC (Java Database Connectivity). Combining NOI for Notes database access with JDBC for relational database access opens up some very interesting possibilities for application developers.

What is JDBC?

The Java Database Connectivity library is a set of Java classes written by JavaSoft (and rewritten by various tool vendors according to the JavaSoft spec) that provide a database independent way of accessing relational data. It provides a relational database object model, and as such may not be appropriate for access to nonrelational databases (though some nonrelational databases do allow you to treat them as relational databases using adapter interfaces such as JDBC, Notes for one, using the NotesSQL ODBC driver). If you're familiar with the ODBC classes (known as LS:DO, for LotusScript:Data Object) that have been shipping with Domino since Release 4.0, you can think of JDBC as a Java API, which more or less accomplishes the same thing.

The Java classes can be found in the package java.sql, and JavaSoft's implementation can be downloaded for free from their Web site (other vendors also give you their implementations for free). The Java classes are really a "driver manager" layer on top of a set of database "drivers," where each driver knows how to access a particular vendor's relational database system (dBase, Informix, Oracle, DB2, Sybase, etc.). Your Java program specifies a URL-like connection string specifying the data source to which you want to connect. The java.sql layer tries to find a driver that is compatible with that data source. If it can find one, it loads it and passes the connection information (database name, and optionally a user name and password) off to that driver, which makes the actual connection. Then SQL queries can be passed through the driver to the back-end database system, possibly over a network, and the results come back and are retrieved via a "result set" class.

This whole system was modeled fairly closely after Microsoft's ODBC (Open Database Connectivity) APIs, which have been in widespread use for years. Notes has provided a set of LotusScript classes for interfacing to ODBC ever since Notes Release 4.0 (the LSXODBC classes, also known as LS:DO, for LotusScript:Data Object).

JavaSoft (and some of the other tool vendors) provide a couple of ways of implementing drivers for individual databases: You can write the entire driver in Java, or you can write a Java driver which in turn passes requests off to an existing ODBC driver. This is known as the *JDBC/ODBC Bridge*. Further information on how the drivers work follows in the next section.

Why Do We Care About JDBC?

A large percentage of the world's data lives in computer-based databases. Production databases at even medium-sized companies routinely grow to sizes exceeding 4 or 5 terabytes (1 terabyte is 1000 gigabytes, or 1,000,000,000,000 bytes). While Notes can in general only sustain good performance on databases in the gigabyte range (plus or minus), it has proven a fantastic tool for "front-ending" serious database applications and workflow applications based on large databases. Data can be accessed on the fly from Notes, and used to populate forms and views. Notes databases can be used as "staging areas" for aggregated data from a back-end database, and from there replicated around the world.

More recently, developers are discovering the possibilities for Web-based applications using Domino. As more companies begin to seriously invest in applications that can be run from a client browser, Domino is proving to be an excellent tool for grabbing data from legacy systems and presenting it on an inter/intranet via HTTP, as well as for collecting user submissions and updating back end storage. Agents play a particularly important role in this kind of system integration, allowing for customized data validation, user authentication, interactivity, and workflow triggering.

I predict that Domino/Notes integration with legacy database systems (relational and others) will be one of the biggest growth areas for Domino application developers, and that JDBC (or its successors) will play a very important role.

Layer Upon Layer

As I stated above, JDBC gives you two options for connecting to a database: the "pure" Java approach, where a vendor has supplied you with a JDBC driver written entirely in Java that talks to the back-end database API; or the JDBC/ODBC Bridge technique, where a small Java driver itself loads and communicates with an existing ODBC driver for the target database.

Of course, the JDBC or JDBC/ODBC driver itself is implemented typically on top of some database vendor's APIs to the real database system. Let's take a look at the actual architecture of a Domino Java NOI Agent that also uses JDBC to retrieve and/or store data in a legacy database system. Figure 13.1 shows the "pure Java" setup, and Figure 13.2 shows the JDBC/ODBC hybrid approach.

Figure 13.1 *Architecture of NOI With Pure Java JDBC.*

Figure 13.2 *Architecture of NOI With JDBC/ODBC Bridge.*

The java.sql classes that make up the Java part are specified by JavaSoft, but any vendor can supply a conforming implementation. Regardless of whose java.sql classes you use, any vendor can supply any of the other components: JDBC drivers, ODBC drivers, or the JDBC/ODBC Bridge. That's the beauty of software API standards (when they work); you can mix and match as you like.

Note that a Domino server that uses ODBC or JDBC to communicate with a back-end database system will usually also include a setup for the database vendor's "client API" software. This is the code that the JDBC or ODBC driver calls to actually talk to the DBMS, which is typically not located on the Domino machine, but elsewhere in the network. It's possible, of course, that a pure Java JDBC driver uses the database system's networking protocol to communicate directly, without going through the Client API layer. ODBC drivers typically do not do that. Ask your vendor how their drivers were implemented.

Which way should you go if you're looking at implementing a Domino/JDBC application? Fortunately you can pick on a per-Agent (or application) basis which driver architecture you will use, because the database URL syntax controls the driver protocol, as we shall see. Here are some points to consider:

- **Performance**. In general, the pure Java JDBC driver will be a better performer than the hybrid JDBC/ODBC architecture, mainly because there is one fewer layer for data to be translated across. This is a rule of thumb, however, not a law graven in stone. Ask your vendor for performance benchmarks, specifically for benchmarks using JDBC. Many driver sets are available for free "on trial" for some amount of time from their vendor. You can write your own benchmark, try out a couple of different vendors' drivers, and make up your own mind, if performance is a key consideration for you.
- **Availability**. There are dozens of choices out there for ODBC drivers for every conceivable database system. There are fewer choices for pure Java JDBC drivers, at least at this time, because the technology is newer. If the target database you want to support does not have an available pure Java driver, then you'll probably have to go with a hybrid JDBC/ODBC solution.
- **Reliability**. A hybrid JDBC/ODBC solution from a vendor that you know and trust could well be better than a pure Java driver from a startup whose code robustness is unproven, or even known to be buggy.

- **Thread safety**. While writing thread safe code is easier in Java than in C or C++, having a JDBC driver written purely in Java is still no guarantee that it is thread safe. As we've seen in earlier chapters of this book, Web-based applications triggered through Domino's HTTP server are generally required to be thread safe. I've shown you how to write thread safe applications and Agents that use NOI, but of course you have no control over the code in a JDBC or ODBC driver you purchase from a third party. Furthermore, you also have no control over the "Client API" layer supplied by the database vendor, if there is one.

The bottom line on this is that you should test the heck out of any JDBC application before deploying it, especially if it needs to be thread safe. Pure Java implementations are not automatically guaranteed to work better than hybrid JDBC/ODBC solutions.

Testing Strategies

Luckily it is possible to do significant testing of multiple vendors' drivers without rewriting lots of software. Because the JDBC interface is standardized (the whole point of it, really), the Agent or application needs to be written only once (but of course you'd want to make sure to test it with a few different back-end databases, if your production environment will require that your code be used that way). Because the connection URL in your Java code accesses a “data source”, not an actual database, by name, the driver (be it JDBC or ODBC) is responsible for resolving the data source name to a real database. Thus you can point the Agent at different databases simply by changing the JDBC or ODBC configuration information. Of course, if you do that you have to be sure that all the databases have the same schema, otherwise your code won't work.

For Web-based applications you can also switch Domino between single- and multithreaded execution of Domino Agents. You can set the **DominoAsynchronizeAgents** environment variable in your server's NOTES.INI file to control this feature (see Chapter 8 for more details). Setting the variable to 1 lets the HTTP server run Agents in a multithreaded way, improving overall throughput. If you set it to 0 (the default), Domino will serialize all Agent execution (only one Agent will run at a time; all others will wait until the currently executing one is done). If you find that your Web-based JDBC or ODBC application is crashing, and you suspect that the culprit might be some code that isn't thread safe, you can set DominoAsynchronizeAgents to 0 (you have to restart the HTTP server for it to take effect). If the crash goes away, then you likely have a thread safety problem.

Installing JDBC and ODBC

Installing the JDBC *driver manager* is easy, or even trivial. This is the code that belongs to the java.sql package, and could well be included in current releases of the Java Development Kit. If so, there's nothing you have to do. If not, you can download the JDBC code from JavaSoft's Web site for free. Then you just need to be sure that the JDBC .class files are on your CLASSPATH.

If you purchase JDBC database drivers from a third-party vendor, you'll need to follow their instructions for installation. Sometimes the drivers also come with the vendor's own java.sql implementation, which you can use instead of JavaSoft's if you wish (check the vendor's installation instructions, they might *require* that to use their drivers you also have to use their driver manager, or they might not).

If you use the JDBC/ODBC bridge, you'll have to make sure that you have the ODBC manager and driver set installed as well. My laptop's Microsoft Windows95 did not come with ODBC, but my desktop's Windows NT had an ODBC manager and a few drivers included. Again, when you purchase a driver set from a vendor it often comes with its own ODBC manager, which when you install the drivers might replace the one on your system. Make sure that your version of JDBC includes a JDBC/ODBC Bridge if you want to use your ODBC drivers (JavaSoft's does).

Whether you're using JDBC alone or in combination with ODBC, you'll have to configure one or more data sources in order to actually access any databases. Think of a data source definition as your system's local name for some remote database. It specifies what driver can be used with the database, the database's "real" name (possibly including remote network information), and so on. This configuration information is used by the Driver Manager to select a driver for the data source whose name you specify in your Java program (if you don't specify a particular driver explicitly), and by the driver to locate the actual database and open a connection to it. Because each vendor's setup procedure is different, I won't detail how to do all that here. Read your vendor's install instructions carefully. For Windows, all the ODBC software I've ever used added an icon to the Control Panel at install time. Clicking on the icon brought up the ODBC Manager belonging to the last one I installed. From there I could select either a driver or a data source to configure, or add a new data source.

To prepare a JDBC example for this chapter I went to the Intersolv Web site (this doesn't constitute a particular endorsement of their software; I chose it because I had used it before and was familiar with it) and downloaded a free trial copy of their DataDirect ODBC manager/driver package, as well as their JDBC/ODBC Bridge software (Intersolv and JavaSoft jointly developed the Bridge). I installed the software, and configured a data source for a *comma delimited* text file that I created as a small sample database (a comma delimited file is one of the database types for which Intersolv supplies an ODBC driver). Apart from the actual download, the setup took about half an hour.

Note that the JDBC/ODBC Bridge does not work with Microsoft's VJ++ development environment. (Microsoft has refused to implement JavaSoft's Java Native Interface - JNI - specification for calling out to C from Java, and JNI is required for the JDBC/ODBC Bridge. The same holds true for the Domino Java NOI, which is why VJ++ can't be used as an NOI development environment.)

JDBC Example

I won't go into deep detail on how JDBC's classes actually work. There are excellent white papers and even a tutorial for JDBC available on the JavaSoft Web site (http://developer.javasoft.com/developer/onlineTraining/jdbc/index.html and http://java.sun.com/products/jdbc). A very good article called "Getting Started With JDBC" by John Papageorge is also available at http://developer.javasoft.com/developer/readAboutJava/jpg/startjdbc.html. The HTML documentation that comes with the software is pretty good too.

Driver and Database Selection

The java.sql.DriverManager is the class responsible for selecting and loading database drivers at connect time. You specify a URL syntax particular to JDBC to tell the DriverManager how to connect to the database you want. The basic layout of a JDBC URL is as follows:

```
jdbc:<subprotocol>:<subname>
```

The first part ("jdbc:") is just the protocol specifier, like "http" or "ftp". The "subprotocol" is the name of either a specific driver (must be a pure Java driver), such as "dbaw" for Symantec's dbANYWHERE, or the special "connectivity mechanism" name "odbc". The latter is used when you are using the JDBC/ODBC Bridge; it tells JDBC to load an ODBC driver specific to the data source name that comes next. The driver manager then has to look up the data source configuration with the ODBC Manager to figure out which driver to load.

The third part of the URL is the data source specification. For an ODBC name, it must be one of the configured data sources. For a Java JDBC driver, it can include a host name and port, such as:

```
jdbc:dbaw://localhost:9988/xyzzy
```

In this case the database server machine name is "localhost", and the requested port number is 9988, followed by the database name. Here's an ODBC example:

```
jdbc:odbc:Employees
```

In this case "Employees" must be a known ODBC data source name.

The JDBC documentation says that the subprotocol can also be a remote naming service, such as "dcenaming." A *remote naming service* is some piece of software that you have access to (locally or over a network) that maps names (in some canonical format) to network addresses. Common naming services include DNS (Domain Naming Service), Novell's NDS (Netware Directory Service), and Sun's NIS (XXX). The JDBC documentation tells you how to register your own subprotocol.

The JDBC style URL is used in the DriverManager.getConnection() call to establish a connection with a database. An instance of the java.sql.Connection class is returned. Once you have a Connection instance you can send off SQL queries in various forms (including parameterized queries), using the java.sql.Statement class. When you execute a query you typically get back an instance of the java.sql.ResultSet class. You use ResultSet methods to iterate through the returned data, usually row by row.

The Code

I created the following simple example using (with permission) a sample program named SimpleSelect included with Intersolv's trial version of their JDBC software. I modified it to use the ODBC data source I have set up on my computer (the source database is a comma delimited text file named TABLE1.TXT and is included on the CD) named Employees. I also modified the logic that writes out the result set, in a vain attempt to make the column headers and the fields in each row line up nicely. I added some new code to also add the data to a Notes database (Ex13.nsf, also on the CD).

I say "vain attempt" because I wasn't particularly successful. This is a good example of where behavior in “ODBC land” is not always the same from driver to driver. The

Intersolv text file driver apparently trims leading zeros from numbers and trailing spaces from strings in a result set (whether you want it to or not). If you develop a nice way of doing that, please let me know.

Listing 13.1 shows you the source code for the simple application I used. It does nothing more than connect to the database, retrieve all rows from a single table, and print out all the rows to the screen. For each row, I create a new Document in the sample Database. In each Document, I use the column name (referred to as the column "header" in the sample code) as the Item name for each column value.

Listing 13.1 JDBC Data Retrieval Example (Ex13Select.java)

```
//
// Copyright:   1990-1996 INTERSOLV, Inc.
//              This software contains confidential and
proprietary
//              information of INTERSOLV, Inc.  You may
study, use, modify
//              and distribute this example for any purpose.
This example
//              is provided WITHOUT WARRANTY either
expressed or implied.

//     This program was modified by Bob Balaban to load the
retrieved data into
//     a Notes database. Modifications Copyright 1997
Looseleaf Software, Inc.
//

import java.net.URL;
import java.sql.*;
import lotus.notes.*;

class Ex13Select
{

     public static void main (String args[])
     {

     // The following code will enable JDBC logging and send
all
     // logging information to a file named 'jdbc.out'

//     try {
//          java.io.OutputStream outFile = new
//                    java.io.FileOutputStream("jdbc.out");
//
//          java.io.PrintStream outStream = new
//                    java.io.PrintStream (outFile, true);
//
//          DriverManager.setLogStream (outStream);
//          }
//     catch (java.io.IOException ex) {
//          System.out.println("Unable to set log stream: "
+
                        ex.getMessage());
//          }

     String url   = "jdbc:odbc:Employees";
     String uid   = "";
     String pwd   = "";

     String query = "SELECT * FROM Table1.txt";
     boolean startedNotes = false;

     try {

          // Create a JDBC driver object from our
          // JDBC to ODBC Bridge in order to register it
          // with the system

          java.sql.Driver d = (java.sql.Driver)
Class.forName (
               "sun.jdbc.odbc.JdbcOdbcDriver").newInstance
();

          // Register the driver
```

```
            DriverManager.registerDriver (d);

            // Attempt to connect to a driver.  Each one
            // of the registered drivers will be loaded until
            // one is found that can process this URL

            Connection con = DriverManager.getConnection (
                           url, uid, pwd);

            // If we were unable to connect, an exception
            // would have been thrown.  So, if we get here,
            // we are successfully connected to the URL

            // Check for, and display and warnings generated
            // by the connect.

            checkForWarning (con.getWarnings ());

            // Get the DatabaseMetaData object and display
            // some information about the connection

            DatabaseMetaData dma = con.getMetaData ();

            System.out.println("\nConnected to " +
  dma.getURL());
            System.out.println("Driver       " +
  dma.getDriverName());
            System.out.println("Version      " +
  dma.getDriverVersion());
            System.out.println("");

            // Create a Statement object so we can submit
            // SQL statements to the driver
            Statement stmt = con.createStatement ();

            // Submit a query, creating a ResultSet object
            ResultSet rs = stmt.executeQuery (query);

            // Display all columns and rows from the result
  set
            // need to init Notes here
```

```
          NotesThread.sinitThread();
          startedNotes = true;
          String dbServ = "";
          String dbName = "book\\Ex13.nsf";
          dispResultSet (rs, dbServ, dbName);

          // Close the result set
          rs.close();

          // Close the statement
          stmt.close();

          // Close the connection
          con.close();
          }
     catch (SQLException ex)
          {
          // A SQLException was generated.  Catch it and
          // display the error information.  Note that there
          // could be multiple error objects chained
          // together
          System.out.println ("\n*** SQLException caught
***\n");

          while (ex != null) {
               System.out.println ("SQLState: " +
ex.getSQLState ());
               System.out.println ("Message:  " +
ex.getMessage ());
               System.out.println ("Vendor:   " +
ex.getErrorCode ());
               ex.printStackTrace();
               ex = ex.getNextException ();
               System.out.println ("");
               }
          }
     catch (java.lang.Exception ex)
          {
            // Got some other type of exception.  Dump it.
            ex.printStackTrace ();
            }

        finally {
                if (startedNotes)
                 NotesThread.stermThread();
              }
        }     // end main

    //------------------------------------------------------------
    ---------
    // checkForWarning
    // Checks for and displays warnings.  Returns true if a
    warning
    // existed
    //------------------------------------------------------------
    ---------

        private static boolean checkForWarning (SQLWarning
    warn)
                throws SQLException
        {
        boolean rc = false;

        // If a SQLWarning object was given, display the
        // warning messages.  Note that there could be
        // multiple warnings chained together

        if (warn != null) {
            System.out.println ("\n *** Warning ***\n");
            rc = true;
            while (warn != null) {
                System.out.println ("SQLState: " +
    warn.getSQLState ());
                System.out.println ("Message:  " +
    warn.getMessage ());
```

```
                System.out.println ("Vendor:   " +
warn.getErrorCode ());
                System.out.println ("");
                warn = warn.getNextWarning ();
                }
           }
           return rc;
      }

      //-------------------------------------------------------------
--------------
      // dispResultSet
      // Displays all columns and rows in the given result
set
      //-------------------------------------------------------------
--------------

     private static void dispResultSet (ResultSet rs, String
dbServ,
                                          String dbName)
           throws SQLException
      {
      int i;
      Session s = null;
      Database db = null;
      Document doc = null;

      // Get the ResultSetMetaData.  This will be used for
      // the column headings
      ResultSetMetaData rsmd = rs.getMetaData ();

      // Get the number of columns in the result set
      int numCols = rsmd.getColumnCount ();
      String headers[] = new String[numCols];

      // Display column headings, save them
      String colHdr;
      for (i=1; i<=numCols; i++)
```

```
            {
             colHdr = rsmd.getColumnLabel(i);
             if (i > 1)
                  System.out.print("  ");
             System.out.print(colHdr);
             headers[i-1] = colHdr;
             }

        System.out.println("");

        // Display data, fetching until end of the result set
        // We'll create a new Document in the Database for each
   row,
        // using the column headers as the Item names.

        try {
            s = Session.newInstance();
            db = s.getDatabase(dbServ, dbName);
             }
        catch (Exception e)
             {
             System.out.println("Couldn't open Notes database "
   + dbName);
             e.printStackTrace();
             s = null;
             db = null;
             }

        boolean more = rs.next ();
        if (!more)
             System.out.println("No rows were retrieved");

        try {
             while (more)
                  {
                  // Loop through each column, getting the
                  // column data and displaying it
                  // create new document
```

```
                    if (db != null)
                        doc = db.createDocument();

                    String value;
                    for (i=1; i<=numCols; i++)
                         {
                         value = rs.getString(i);
                         if (i > 1) System.out.print("  ");
                         System.out.print(value);

                         // add the value to the document
                             doc.appendItemValue(headers[i-1],
value);
                         } // end for

                    System.out.println("");

                    // Fetch the next result set row, save the
document
                    more = rs.next ();
                    doc.save(true, false);
                    }     // end while
               }  // end try
          catch (Exception e) { e.printStackTrace(); }

          }  // end dispResultSet
     }  // end class
```

Figure 13.3 shows the output to the command window, and Figure 13.4 shows the resulting records in the Notes database's view.

Figure 13.3 *Ex13Select output to command window.*

Figure 13.4 *Ex13 Database view.*

Discussion of Data Retrieval Example

Note that this is a single-threaded example, so we have to do the static method Notes initialization thing before we call the subroutine (dispResultSet) that creates the Notes

Documents. The NotesThread.stermThread() call is in the last *finally* clause of the main program. We had to also use a boolean to tell us in the *finally* whether or not Notes was ever actually initialized, because an exception occurring before the init call would always end up in the *finally* logic, and if we try to terminate a Notes thread that was never initialized, there'll be trouble. Another way of doing that would be to use nested *try* blocks, with a *finally* on the inner one where the NotesThread.sinitThread() call is made.

Other than that there isn't too much to say about this example. The usage of the JDBC classes is quite ordinary and straightforward. You (especially you SQL heads out there) should be able to follow it easily.

The only slightly out of the ordinary thing about it is the registration of the JDBC/ODBC Bridge class (sun.jdbc.odbc.JdbcOdbcDriver), using the Class.forName() call to load the .class file into the Java VM. The Class instance returned from forName() is used to create an actual instance of the class in the newInstance() call. That instance is passed to DriverManager.registerDriver(), another static method invocation. The driver is registered as the handler for the "odbc" subprotocol, so that when our URL ("jdbc:odbc:Employees") is processed by the DriverManager.getConnection() call, the correct driver is used.

You've seen me state elsewhere in this book that I prefer to avoid using classes from the "sun" package, and that I like to stick to classes in the java or javax (or lotus.notes) packages. Why, then, are we using sun.jdbc.odbc.JdbcOdbcDriver? Because this is a vendor-specific software configuration, that is, the JDBC/ODBC driver implementation is not "standard" in the way that the classes in java.lang or javax.servlet are. And the reason that we use Class.forName() to load the bridge class instead of just saying *new jdbc.odbc.JdbcOdbcDriver()* is that we want to defer the class lookup on the computer to run time. Because the class is nonstandard, we can't be sure that every machine will have it lying around. Thus we would rather not have the class loaded when the program is loaded, but when the Class.forName() call is actually executed. This gives us an opportunity (of which I have, alas, not availed myself in this particular example) to do better error recovery.

A Word on Agent Security

As we've discussed elsewhere in this book, Agents come in two flavors: restricted and unrestricted. Restricted Agents cannot make use of as many server resources as can unrestricted Agents. One of the things restricted Agents can't do is load DLLs (other than the Notes DLL which implements NOI, of course). Because "pure" JDBC only causes Java classes to be loaded, there are no problems with security. Restricted Agents, however, will not be able to use the hybrid JDBC/ODBC Bridge configuration as virtually all ODBC drivers are implemented in C or C++.

You might want to consider this limitation if you're implementing background Agents that need to use JDBC.

Summary

Relational database connectivity for Notes/Domino is, in my opinion, both one of the most powerful and also the most often overlooked capabilities among groupware developers. Combining NOI with JDBC gives you the power to move all kinds of data between Notes databases and any other database system that supports SQL queries.

You can, for example, write a multithreaded Java Agent that is invoked from a browser via the Domino HTTP server, which collects user input using a simple Domino form. The Agent can then perform multiple queries on a remote database system, format the results as HTML output, and send it back to the browser, possibly updating a Notes database as well. Updating the Notes database can, in turn, trigger a workflow application, such as serial approval, email notification, inventory restocking, or almost anything else.

I'd say that the only area of database connectivity where Domino/NOI is weak involves moving around binary data (sometimes called *BLOBs*, for *Binary Large OBjects*). The NOI object model does not (yet) support binary data (and neither do lots of JDBC/ODBC drivers, by the way). As this kind of functionality becomes more popular (especially for Web-based applications such as video streaming), I'm sure you'll see it added to NOI, making it even more powerful.

Domino's ability to use Java to integrate these various back-end tools is unmatched by any other product.

Coming up in Chapter 14 is a prognostication of where some of the NOI-related technologies could go (and we hope will go) with the next release of Domino.