Java Reference Guide

Hosted by Steven Haines

Java Database Connectivity (JDBC) API

Last updated Mar 14, 2003.

The Java Database Connectivity (JDBC) API provides an interface to access a wide range of data sources, including Structured Query Language (SQL) based databases all the way down to flat files (text files.) Because of its acceptance throughout the Java industry, the JDBC API has become part of the core Java API; it is now at version 3.0.

Several years back, Microsoft provided a data interchange standard, called the Open Database Connectivity (ODBC) interface, that provided application developers wishing to use databases with an invaluable service. Before an open standard was developed, an application developer had to either obtain source code libraries to talk to specific database systems, or write source code that knew how to talk to each and every database the application could support. With the advent of the ODBC interface, database developers could create generic ODBC drivers to talk to their respective databases, and then application developers' code talked to ODBC, which routed the request.

Configuring an application to work with a database is simple: the user installs a database and sets up a data source (a name that ODBC uses to find the database), then points the database application to that data source.

Java extends the ODBC model. The Java Database Connectivity (JDBC) model supports ODBC-based databases and provides a truly platform-independent database access model.

JDBC has four primary pieces, used for each database access phase:

- DriverManager: the `DriverManager` class loads and configures a database driver on your client
- Connection: the `Connection` class performs connection and authentication to a database server
- Statement / PreparedStatement: the `Statement` and `PreparedStatement` classes send SQL statements to the database engine for preprocessing and eventually execution
- ResultSet: the `ResultSet` class allows for the inspection of results from Statement executions

At the core of JDBC is the Structured Query Language (SQL). InformIT has a plethora of articles describing how to write and tune SQL statements, so I will refer you to those resources for more guidance.

The first step in using JDBC is acquiring a driver for your database. Your database probably came with a JDBC driver or an ODBC driver, maybe both. If the database already has its JDBC driver you are set. Otherwise, you can use an ODBC-JDBC bridge to make use of the ODBC drivers; note that there are extreme performance degradations in using the ODBC-JDBC bridge. Furthermore, you can use a third party JDBC driver written specifically for your database. You can find links to JDBC drivers at Sun Microsystems Web site at:

http://servlet.java.sun.com/products/jdbc/drivers

JDBC drivers come in four types (numbered from 1 to 4):

1. JDBC-ODBC Bridge plus ODBC Driver: provides a JDBC interface to an ODBC driver
2. Native API-partly Java driver: converts JDBC calls into calls to the client API for your specific

database; requires some native code on each OS you deploy to

3. JDBC-Net pure Java driver: JDBC calls are sent in a database independent format to Java code running on the database server that in turn translates that request to database specific code
4. Native-protocol pure Java driver: Converts JDBC calls directly to database network calls

Type 3 and 4 drivers are preferred; types 1 and 2 act as place holders for databases that do not yet have a full Java implementation. Once you have this `DriverManager` you use it to establish a connection with the database through its getConnection() method. The getConnection() method prototypes are defined as follows:

```
getConnection(String url)
getConnection(String url, Properties info)
getConnection(String url, String user, String password)
```

The URL `String` is a database vendor-specific string that tells the JDBC driver what database to connect to and how to establish that connection. The getConnection() method returns a `Connection` object. For example, if you were using an Oracle database, the connection would look something like the following:

```
Connection conn = DriverManager.getConnection(
"jdbc:oracle:thin:@mydbserver:1521:mysid",
                        "scott", "tiger" );
```

Where the URL is composed of the main protocol "jdbc", the sub-protocol "oracle:thin", the host name "mydbserver", the port "1521", and the database SID name "mysid". The connection is made as user name "scott" and password "tiger". Again, the JDBC URLs are driver-specific; this example used the Oracle thin driver (type 4 driver) that ships with Oracle.

`Connection` is an interface that the JDBC specific driver implements. The primary functionality that a Connection provides you as an application developer is the ability to create `Statements` and `PreparedStatements`.

A `Statement` is used to send a query to a database: the `Statement` is compiled into a format to send to the database, it is sent to the database, its result is obtained, and the `Statement` is discarded.

A `PreparedStatement` follows the same steps the first time it is executed, but then the statement is saved for subsequent use in its compiled state. The `PreparedStatement`'s parameters are specified as variables; it is storing the compiled structure of the statement and not the statement itself. As a result, you can fill in the parameters when using the statement, so that its use is specific to your requirements.

A `Statement` can be created by calling the `Connection` class's `createStatement()` method:

```
Statement createStatement()
Statement createStatement(int resultSetType, int resultSetConcurrency)
```

You can create a generic `Statement` object or one that will generate ResultSet objects with the given type and concurrency. Similarly, you create a `PreparedStatement` by calling the `Connection` class's `prepareStatement()` method:

```
PreparedStatement prepareStatement(String sql)
PreparedStatement prepareStatement(String sql,
➥int resultSetType, int resultSetConcurrency)
```

You can create a `PreparedStatement` to service a specified SQL statement and, optionally, can specify the type of `ResultSet` to be generated. The following example creates a `PreparedStatement`:

```
try {
  PreparedStatement preparedStatement = conn.prepareStatement(
    "SELECT * FROM HomePhoneNumbers WHERE name = ?" );
}
catch ( SQLException e ) {
  e.printStackTrace();
}
```

Again, `PreparedStatement` is an interface that the JDBC specific driver implements. Once you have built the `PreparedStatement`, you can execute the query using one of the following execute methods:

```
boolean execute()       Executes any kind of SQL statement
ResultSet executeQuery()   Executes the SQL query in this
PreparedStatement
 ➥object and returns the result set generated by the query
int executeUpdate()      Executes the SQL INSERT, UPDATE or
 ➥DELETE statement in this PreparedStatement object
```

To query a database, you call the executeQuery() method, which returns an object that implements the `ResultSet` interface. The following shows how you would execute the aforementioned query:

```
ResultSet rs = preparedStatement.executeQuery();
```

Again, the `ResultSet` interface is implemented by one of the JDBC specific driver classes. The `ResultSet` interface is quite large, but the navigation through rows in the table is straightforward:

```
boolean first()    JDBC 2.0 Moves the cursor to the first row in the
result set.
boolean isFirst()   JDBC 2.0 Indicates whether the cursor is on the first
row
➥of the result set.
boolean isLast()   JDBC 2.0 Indicates whether the cursor is on the last
row of the result set.
boolean last()    JDBC 2.0 Moves the cursor to the last row in the result
set.
void moveToCurrentRow()  JDBC 2.0 Moves the cursor to the remembered
cursor position,
➥ usually the current row.
void moveToInsertRow()   JDBC 2.0 Moves the cursor to the insert row.
boolean next()   Moves the cursor down one row from its current position.
boolean previous()     JDBC 2.0 Moves the cursor to the previous row in
the result set.
```

Once you have the `ResultSet` object initialized, you can traverse the list using the following syntax:

```
while( rs.next() ) {
    // Do stuff with the data
}
```

Now the `ResultSet` interface defines a set of methods referred to as the getXXX methods by the JDBC community. These methods are used for retrieving specified datatypes from either a named or indexed column in the current row. You can retrieve data of various types, from integers to arrays to strings.

Listing 1 puts this together into a complete example.

Listing 1. JDBCExample.java

```
package com.informit.jdbc;

import java.sql.*;

public class JDBCExample {
  public static void main( String[] args ) {
    try {
      DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
      Connection conn = DriverManager.getConnection(
➥"jdbc:oracle:thin:@mydbserver:1521:mysid",
                           "scott", "tiger" );

      // Create a Statement
      PreparedStatement ps = conn.prepareStatement(
➥"SELECT state FROM HomePhoneNumbers WHERE name = ?" );
      ps.setString( 1, "Steve" );

      ResultSet rs = ps.executeQuery();
      // Iterate through the result and print the employee names
      while (rs.next ()) {
       System.out.println( "State: " + rs.getString( "state" ) );
      }
    }
    catch( Exception e ) {
      e.printStackTrace();
    }
  }
}
```

Summary

JDBC is the Java interface to interacting with databases; it provides a simple database-agnostic programming paradigm to access SQL-based databases. Inevitiably, in your development efforts you will to need to read something from or write something to a database. I encourage you to follow up this general introduction with many of the resources we have available here at InformIT.com.

Save To Your Account   < Back   Page 96 of 650   Next >