Writing JDBC Applications with MySQL

*CS442, Lab Session*

## Table of Contents

* [Before Lab](#_gjdgxs)
* [During Lab](#_1fob9te)
  + [Database schema and instances in use](#_3znysh7)
  + [Your task](#_2et92p0)
  + [What�s in the skeleton code?](#_tyjcwt)
  + [Your to-do list](#_2s8eyo1)
* [Resources](#_17dp8vu)

## Before Lab

You can write MySQL applications in a variety of languages. The languages that most people use with MySQL are PHP and Perl, but a sometimes overlooked option is the MySQL Connector/J driver, which allows you to develop Java applications that interact with your MySQL server.

JDBC is based on an approach similar to that used in the design of Perl and Ruby DBI modules, Python's DB-API module, and PHP's PEAR::DB class. This approach uses a two-tier architecture:

* The top level is visible to application programs and presents an abstract interface for connecting to and using database engines. The application interface does not depend on details specific to particular engines.
* The lower level consists of drivers for individual database engines. Each driver handles the details necessary to map the abstract application interface onto operations that a specific engine will understand.

The JDBC interface allows developers to write applications that can be used with different databases with a minimum of porting effort. Once a driver for a given server engine is installed, JDBC applications can communicate with any server of that type. By using MySQL Connector/J, your Java programs can access MySQL databases.

To use Java applications with MySQL, you may need to install some additional software:

* **Java compiler**: If you want to compile and run Java programs, you'll need a Java compiler (such as *javac* or *jikes*) and a runtime environment. If these are not already installed on your system, you can get them by obtaining a Java Software Development Kit (SDK) from [*java.sun.com*](http://java.sun.com/). I use Eclipse, an open-source Java IDE. You can download it from <http://www.eclipse.org/>.
* **JRE/JDK**: If you want only to run precompiled applications, no compiler is necessary, but you'll still need a Java Runtime Environment (JRE). This too may be obtained from [*java.sun.com*](http://java.sun.com/). If you use Eclipse, you need to download JDK. I use JDK 6.
* **MySQL Connector/J:�** MySQL Connector/J works within the framework of the Java JDBC interface, an API that allows Java programs to use database servers in a portable way. I use MySQL connector 5.1.22 (<http://dev.mysql.com/downloads/connector/j/>)
* **MySQL:** You can download from <http://dev.mysql.com/downloads/>.� Remember the password you enter when you configure MySQL. It will be the password (for root user) needed for [connecting to MySQL Server](#_3dy6vkm).

After successfully installed the software, you need to setup the paths of the software. I assume you know how to do that from your previous Java programming experience.

**I also provide** [**the skeleton code**](http://www.cs.stevens.edu/~hwang4/courses/cs442/lab/example.java) **that you will need to use in the lab.� Please download it and make sure it can be compiled successfully on your machine.� Running the code� should return the following message:**

Connecting to database...

Creating database...

Database created successfully...

branch id = 10, name = Main, address = 1234 Main St.

branch id = 20, name = Richmond, address = 23 No.3 Road

2 rows were retrieved

Goodbye!

If you have trouble compiling, double check that you have a Java Software Development Kit installed and make sure that the MySQL Connector/J driver is listed in your CLASSPATH environment variable.

## During Lab

## �•••••• Database schema & instances in use

## In the lab session, you will use JDBC to implement a database for Vancouver Motor Vehicle Office. The database is named *VehicleOffice*. It consists of four tables:

�•••••••• branch(branch\_id integer, branch\_name varchar(20), branch\_addr varchar(50), branch\_city varchar(20), branch\_phone integer);

�•••••••• driver(driver\_ssn integer, driver\_name varchar(20), driver\_addr varchar(50), driver\_city varchar(20), driver\_birthdate date, driver\_phone integer);

�•••••••• license(license\_no integer, driver\_ssn integer, license\_type char, license\_class integer, license\_expiry date, issue\_date date, branch\_id integer);

�•••••••• exam(driver\_ssn integer, branch\_id integer, exam\_date date, exam\_type char, exam\_score integer);

## And we have the following instances of these four tables:

## Branch:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Branch\_id | Branch\_name | Branch\_addr | Branch\_city | Branch\_phone |
| 10 | Main | 1234 Main St. | Vancouver | 5551234 |
| 20 | Richmond | 23 No.3 Road | Richmond | 5552331 |
| 30 | West Creek | 251 Creek Rd. | Sechelt | 5552511 |
| 40 | Blenheim | 1342 W.22 Ave. | Burnaby | 5551342 |

## Driver:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Driver\_ssn | Driver\_name | Driver\_addr | Driver\_city | Driver\_birthdate | Driver\_phone |
| 11111111 | Bob Smith | 111 E.11 st. | Vancouver | 1975-01-01 | 5551111 |
| 22222222 | John Walters | 222 E.22 St. | Burnaby | 1976-02-02 | 5552222 |
| 33333333 | Troy Rops | 333 W.33 Ave | Richmond | 1970-03-03 | 5553333 |
| 44444444 | Kevin Mark | 444 E.4 Ave. | Vancouver | 1974-04-04 | 5554444 |

## License:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| License\_no | Driver\_ssn | Licence\_type | License\_class | License\_expiry | Issue\_date | Branch\_id |
| 1 | 11111111 | D | 5 | 1999-05-25 | 1997-05-25 | 20 |
| 2 | 22222222 | D | 5 | 1998-08-29 | 1996-08-29 | 40 |
| 3 | 33333333 | L | 5 | 1997-12-27 | 1997-06-27 | 20 |
| 4 | 44444444 | D | 5 | 1999-08-30 | 1997-08-30 | 40 |

## Exam:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Driver\_ssn | Branch\_id | Exam\_date | Exam\_type | Exam\_score |
| 11111111 | 20 | 1997-05-25 | D | 79 |
| 11111111 | 20 | 1997-12-02 | L | 97 |
| 22222222 | 30 | 1996-05-06 | L | 25 |
| 22222222 | 40 | 1996-06-10 | L | 51 |
| 22222222 | 40 | 1996-08-29 | D | 81 |
| 33333333 | 10 | 1997-07-07 | L | 45 |
| 33333333 | 20 | 1997-06-27 | L | 49 |
| 33333333 | 20 | 1997-07-27 | L | 61 |
| 44444444 | 10 | 1997-07-27 | L | 71 |
| 44444444 | 20 | 1997-08-30 | D | 65 |

## �•••••• Your tasks at glimpse

## Your tasks include:

## (1)   Create the schema of the tables;

## (2)   Input the data to the tables;

## (3)   Apply queries on the tables.

## More details of your tasks can be found at [your to-do list](#_2s8eyo1).

## �•••••• What�s in the skeleton code

## The skeleton code provides sample code of how to use JDBC for database applications.� It has the following steps.

## Step 1&2: Connecting to the MySQL Server

To connect to the MySQL server, register the JDBC driver you plan to use, then invoke its getConnection() method. The following short program, *Connect.java*, shows how to connect to and disconnect from a server running on the local host. It accesses a database named test, using a MySQL account with a user name root and password of root (This is my setup of MySQL. You may have different username and password):

**import** java.sql.\*;

**public** **class** example {

�� // JDBC driver name and database URL

�� **static** **final** String *JDBC\_DRIVER* = "com.mysql.jdbc.Driver";�

�� **static** **final** String *DB\_URL* = "jdbc:mysql://localhost/test";

�� //� Database credentials

�� **static** **final** String *USER* = "root";

�� //the user name;

�� **static** **final** String *PASS* = "root";

�� //the password;

��

�� **public** **static** **void** main(String[] args) {

�� Connection conn = **null**;

�� Statement stmt = **null**;

�� **try**{

����� //STEP 1: Register JDBC driver

����� Class.*forName*("com.mysql.jdbc.Driver");

����� //STEP 2: Open a connection

����� System.*out*.println("Connecting to database...");

����� conn = DriverManager.*getConnection*(*DB\_URL*, *USER*, *PASS*);

����� ��

�� }catch (Exception e)

���������� {

�������������� System.err.println ("Cannot connect to database server");

���������� }

���������� finally

���������� {

�������������� if (conn != null)

�������������� {

������������������ try

������������������ {

���������������������� conn.close ();

�������������������� ��System.out.println ("Database connection terminated");

������������������ }

������������������ catch (Exception e) { /\* ignore close errors \*/ }

�������������� }

���������� }

������ }

�� }

The arguments to getConnection() are the connection URL and the user name and password of a MySQL account. As illustrated, JDBC URLs for MySQL consist of jdbc:mysql:// followed by the name of the MySQL server host and the database name. An alternate syntax for specifying the user and password is to add them as parameters to the end of the connection URL:

�� jdbc:mysql://localhost/test?user=root&password=root

When you specify a URL using this second format, getConnection() requires only one argument. For example, the code for connecting to the MySQL server in *Example.java* could have been written like this:

�� String userName = "root";

�� String password = "root";

�� String url = "jdbc:mysql://localhost/test?user="

������������������ + userName

������������������ + "&password="

������������������ + password;

�� Class.forName ("com.mysql.jdbc.Driver").newInstance ();

�� conn = DriverManager.getConnection (url);

getConnect() returns a Connection object that may be used to interact with MySQL by issuing queries and retrieving their results. (The next section describes how to do this.) When you're done with the connection, invoke its close() method to disconnect from the MySQL server.

To increase the portability of your applications, you can store the connection parameters (host, database, user name, and password) in a Java properties file and read the properties at runtime. Then they need not be listed in the program itself. This allows you to change the server to which the program connects by editing the properties file, rather than by having to recompile the program.

## Step 3 � 7: Issuing SQL Queries

To process SQL statements in a JDBC-based application, create a Statement object from your Connection object. Statement objects support an executeUpdate() method for issuing queries. The queries can be the SQL statements that

***(1)   Step 3: create the database***:

String sql = "CREATE DATABASE VehicleOffice";

����� stmt.executeUpdate(sql);

***(2)   Step 4: Specify the current database to use:***

����� sql = "use VehicleOffice";

����� stmt.executeUpdate(sql);

***(3)   Step 5: Create tables in the current database:***

sql = "create table branch( branch\_id integer not null PRIMARY KEY, " +

����� ���������� "branch\_name varchar(20) not null," +

����� ���������� "branch\_addr varchar(50)," +

����� ���������� "branch\_city varchar(20) not null," +

����� ���������� "branch\_phone integer)";

stmt.executeUpdate(sql);

***(4)   Step 6: Insert data into tables:***

sql = "insert into branch values(10, 'Main', '1234 Main St.', 'Vancouver', 5551234)";

stmt.executeUpdate(sql);

***(5)   Step 7: Retrieve data from the tables:***

For statements such as SELECT queries that retrieve information from the database, use executeQuery(). After calling this method, create a ResultSet object and use it to iterate through the rows returned by your query. The following example shows one way to retrieve the contents of the branch table:

�� Statement s = conn.createStatement ();

�� s.executeQuery ("SELECT branch\_id, branch\_name, branch\_addr FROM branch");

�� ResultSet rs = s.getResultSet ();

�� int count = 0;

�� while (rs.next ())

�� {

������ int idVal = rs.getInt ("branch\_id");

������ String nameVal = rs.getString ("branch\_name");

������ String addrVal = rs.getString ("branch\_addr");

������ System.out.println (

�������������� "id = " + idVal

�������������� + ", name = " + nameVal

�������������� + ", address = " + addrVal);

������ ++count;

�� }

�� rs.close ();

�� s.close ();

�� System.out.println (count + " rows were retrieved");

executeQuery() does not return a row count, so if you want to know how many rows a result set contains, you should count them yourself as you fetch them.

To obtain the column values from each row, invoke get*XXX*() methods that match the column data types. The getInt() and getString() methods used in the preceding example return integer and string values. As the example shows, these methods may be called using the name of a result set column. You can also fetch values by position. For the result set retrieved by the SELECT query in the example, id, name, and category are at column positions 1, 2 and 3 and thus could have been obtained like this:

�� int idVal = rs.getInt (1);

�� String nameVal = rs.getString (2);

�� String catVal = rs.getString (3);

ResultSet objects, like Statement objects, should be closed when you're done with them.

To check whether or not a column value is NULL, invoke the result set object's wasNull() method after fetching the value. For example, you could check for a NULL value in the name column like this:

�� String nameVal = rs.getString ("name");

�� if (rs.wasNull ())

������ nameVal = "(no name available)";

### Error Handling

If you want to trap errors, execute your JDBC operations within a try block and use an exception handler to display information about the cause of any problems that occur. JDBC provides getMessage() and getErrorCode() methods that may be invoked when an exception occurs to obtain the error message and the numeric error code. The following example deliberately issues a malformed query. When it runs, the executeQuery() method fails and raises an exception that is handled in the catch block:

�� try

�� {

������ Statement s = conn.createStatement ();

������ s.executeQuery ("XYZ"); // issue invalid query

������ s.close ();

�� }

�� catch (SQLException e)

�� {

������ System.err.println ("Error message: " + e.getMessage ());

������ System.err.println ("Error number: " + e.getErrorCode ());

�� }

## �•••••• Your to-do list

To finish this lab session, you need to do the following tasks.

**� Task 1 & 2**:� The skeleton code only creates the table branch and driver, you need to create the tables license and exam.

**� Task 3 & 4**: The skeleton code only inserts part of records into the table branch and driver, you need to insert the remaining ones into these two tables.

� **Task 5 & 6**: Enter the tuples of the tables license and exam.

� **Task 7**: Display the answer of the following queries:

�•••••••• Q1:� Find the names of the drivers who got the license from the branch �Richmond�.

�•••••••• Q2:� Find the names of the drivers who took at least 2 exams at the same branch.

�•••••••• Q3:� Find the names of the drivers whose exam scores get lower when he/she took more exams.

## �•••••• At the End of Lab

You demonstrate your code and the answers of Q1, Q2 and Q3 to the instructor or any of the three course assistants (CAs) by the end of Oct 28�s lab session.

**��� Grade Based On:**

�•• Construction of 2 tables (20%, 10% per table)

�•• Tuple insertion of 4 tables (40%, 10% per table)

�•• Correct implementation of 3 queries (40%, 10% for the first two queries each, 20% for the 3rd query)

## Resources

* Sun's Java site is a clearinghouse for all kinds of Java-related information:� [http://java.sun.com/:](http://java.sun.com/:%20) You can obtain the Java Software Development Kit or Java Runtime Environment here. The specification for the JDBC API is also available on this site, should you wish to read more about it.

�•••••••• Introductory articles describing other MySQL APIs based on architecture similar to that used for JDBC may be found at [http://www.kitebird.com/articles/.](http://www.kitebird.com/articles/.%20) APIs discussed in the articles include Perl DBI, PHP PEAR::DB, Python DB-API, and Ruby DBI.