Introduction to Java Database Connectivity (JDBC) Technology

Table of Contents

Getting Started with JDBC
  What is JDBC?
  Creating a Database
  Installing JDBC and a JDBC Driver
Establishing JDBC Connection
  Loading JDBC Drivers
  Connecting to a Database
JDBC Statements
  Creating Statements
  Inserting Data into a Database
  Retrieving Data from a Database
  Retrieving Values from Result Sets

Bibliography

Getting Started with JDBC

What is JDBC?

Java Database Connectivity (JDBC) is a Java API that allows programmers to access and work with relational database management systems from the Java programming language. The current version of the JDBC API [JDBCAPI] is 3.0 and it is included into the Java 2 Standard Edition, version 1.4.x [J2SE].

The JDBC API 3.0 consists of two Java packages: java.sql and javax.sql. The first package implements the so-called JDBC Core API, whereas the second package is referred to as the JDBC Optional Package API. Technically, the java.sql package is an extension of the java.sql package from a client-side API to a server-side API. Thus, for developing server-side extensions programmers usually work with the javax.sql package.

Basically, the JDBC API is a software library providing a number of Java classes and interfaces that allows programmers to:

- Establish a connection to a wide range of different Relational Database Management Systems (RDBMS). Note, that each major RDBMS, such as Oracle, Microsoft SQL, or MySQL is supported.
- Access a database within a particular system.
- Manipulate the structure of a database by creating, deleting, or altering relations from that database.
Manipulate the content of a database by inserting, deleting, or updating records in relations from the database.
Retrieve the content from a database by querying the database.

Creating a Database

To demonstrate the basic capabilities of JDBC we will work with a simple test database. The following example can be used for that purpose. Suppose we want to develop a simple database application for a small shop, which sells computer components. The shop manages a small database keeping the data about its customers, products and sales. Thus, the database consists of the following domains and relations.

Example 1. Shop Database

Domains

<table>
<thead>
<tr>
<th>Domain Name</th>
<th>Domain Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cid</td>
<td>Integer</td>
</tr>
<tr>
<td>cname</td>
<td>String</td>
</tr>
<tr>
<td>ccity</td>
<td>String</td>
</tr>
<tr>
<td>cphone</td>
<td>Integer</td>
</tr>
<tr>
<td>pid</td>
<td>Integer</td>
</tr>
<tr>
<td>pname</td>
<td>String</td>
</tr>
<tr>
<td>pprice</td>
<td>Integer</td>
</tr>
<tr>
<td>tdate</td>
<td>Date</td>
</tr>
<tr>
<td>tqnt</td>
<td>Integer</td>
</tr>
</tbody>
</table>

Relations

<table>
<thead>
<tr>
<th>Relation Name</th>
<th>Relation Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>(cname, ccity, cphone, cid)</td>
</tr>
<tr>
<td>Product</td>
<td>(pname, pprice, pid)</td>
</tr>
<tr>
<td>Transaction</td>
<td>(cid, pid, tdate, tqnt)</td>
</tr>
</tbody>
</table>

The database schema will be implemented by means of the open source relational database management system MySQL [MySQL]. Here we won't go into details of a MySQL installation or how to implement a database schema in MySQL. We will suppose that we have a MySQL server running with the above database schema implemented in a database with the name "example". For accessing the "example" database on the MySQL server the username "student" with the password "student" has been created. This user account has the rights to insert, update, and retrieve the data from the database.

Installing JDBC and a JDBC Driver
The latest 3.0 version of JDBC is included in Java 2 Standard Edition, version 1.4.x. For each particular RDBMS there is a special JDBC driver, which needs to be installed. Usually, JDBC drivers are installed by following instructions provided by the driver vendor. In most cases installing JDBC drivers is as simple as copying the compressed Java Archive (JAR) file containing the driver to a specific directory on your operating system. Additionally, you will need to include the driver JAR file into the CLASSPATH environment variable.

The JDBC driver for MySQL is called Connector/J [ConnectorJ]. The current version is 3.0.9. To set the CLASSPATH variable to point to the Connector/J driver on the Linux operating system you need to type something like this:

**Example 2. Installing Connector/J**

```bash
# installation in directory /mysql/connector-j
cp mysql-connector-java-3.0.9-stable.jar /mysql/connector-j

# set the CLASSPATH
export CLASSPATH=$CLASSPATH:/mysql/connector-j/mysql-connector-java-3.0.9-stable.jar
```

On the Windows operating system you can set the CLASSPATH environment variable from the Control Panel.

**Establishing JDBC Connection**

Before we can start with manipulating the content of a database with JDBC we need to establish a connection to that database. Usually, establishing a connection is accomplished in two steps. Firstly, we need to load the appropriate JDBC driver for the RDBMS that we have. At the next step we open a connection to a specific database managed by the system.

**Loading JDBC Drivers**

Loading a JDBC driver is very simple and involves just a single line of Java code.

**Example 3. Loading a JDBC Driver**

```java
...  
try{
    Class.forName("com.mysql.jdbc.Driver");
}catch(ClassNotFoundException exc){
    exc.printStackTrace();
}
...
```
This line of code will tell the DriverManager which Java class to load as a JDBC driver class. If the specified class cannot be found at the CLASSPATH then an exception will be thrown. The name of the driver class can be found in the documentation of a particular JDBC driver. In the above example we load the JDBC driver for MySQL RDBMS.

The complete code of the ConnectionManager class, where the MySQL JDBC driver is loaded is accessible online at http://coronet.iicm.edu/shop/src/shop/db/ConnectionManager.java.

Connecting to a Database

The next step in establishing a connection to the "example" database is to instruct the loaded driver to connect to the RDBMS. Again, this can be accomplished by a single line of Java code:

**Example 4. Loading a JDBC Driver**

```java
...
Connection connection_;  
...
try{
    connection_ = DriverManager.getConnection("jdbc:mysql://" + host + "/" + db, username, password);
}catch(SQLException exc){
  exc.printStackTrace();
}
... 
```

In the above example two new Java classes from the JDBC API are introduced. The first class is the Connection class. An instance of this class represent a single connection to a particular database managed by the system. The second class is the DriverManager class, which provides a number of static class methods for managing JDBC drivers, obtaining connections, setting parameters of particular connections, and so on.

To create a particular connection to a database we call the getConnection() method of the DriverManager class. This method takes three arguments:

- A JDBC URL, which follows the standard URL syntax. The protocol part starts always with "jdbc:" followed by the name of the RDBMS (in our case "mysql") and terminated with "://" symbols. Thus, the protocol part in our example is "jdbc:mysql://". The host part identifies the host where the RDBMS is running. In the case that the Java application connecting to the RDBMS and the system itself are running on the same machine "localhost" can be used to identify the host. Otherwise a fully qualified DNS name must be used, such as "coronet.iicm.edu". Finally, the name of a particular database must be supplied preceeded with the slash character. In our case this would be "/example".
• A registered username that has the proper privileges for manipulating the database.
• Corresponding password for the username.

Note, that a SQLException is thrown if the DriverManager can not connect to the database for whatever reasons. For example, if the provided database is not existent, or the user credentials are not correct. The SQLException is a part of the JDBC API.

Again, the complete code of the ConnectionManager class, implementing the above functionality can be found online at http://coronet.iicm.edu/shop/src/shop/db/ConnectionManager.java.

**JDBC Statements**

In order to insert, update, or retrieve data from a database programmers need to work with instances of the Statement class. The Statement class provides a number of public methods for executing different commands in the database. Depending on the type of a particular command the appropriate method should be invoked. For instance, to insert some data into a relation in a database, the executeUpdate() method should be invoked on an instance of the Statement class. Similarly, to retrieve data from a database, the executeQuery() method should be invoked.

**Creating Statements**

JDBC statements are created by an instance of the Connection class. Thus, the Connection instance that we created in the previous example can be used to obtain instances of the Statement class.

**Example 5. Creating a JDBC Statement**

```java
try{
    Statement statement = connection_.createStatement();
}catch(SQLException exc){
    exc.printStackTrace();
}
```

If an error occurs during the execution of the createStatement() method a SQLException will be thrown.

The complete code of the Persistent class, implementing the above functionality can be found online at http://coronet.iicm.edu/shop/src/shop/db/Persistent.java.

**Inserting Data into a Database**
The Statement class provides a public interface with numerous methods. Depending on the type of the command that we are submitting to a database we need to choose the appropriate method and invoke it on an instance of the Statement class. Thus, for inserting data into a relation of a particular database we need obviously to invoke one of the executeUpdate() methods. Note, that there exist a number of different executeUpdate() methods, depending on arguments of the command that we are submitting to the database. For example, if we need to retrieve the primary keys, which are generated automatically by the system we need to invoke a special executeUpdate method. However, if we are not interested in these automatically generated keys, or there are no keys that are generated automatically we can invoke the most simple executeUpdate method.

Example 6. Inserting Data into a Relation

```java
... try{
    String insert_sql_stmt = "INSERT INTO " + table + " VALUES(" + values + ");"
    int row = statement.executeUpdate(insert_sql_stmt,
            Statement.RETURN_GENERATED_KEYS);
    ResultSet keys = statement.getGeneratedKeys();
} catch(SQLException exc){
    exc.printStackTrace();
}
...```

First of all, we need to create the corresponding SQL command that will insert the data. Suppose that the variables "table" and "values" refer to the name of a table in the database and the values that we want to insert into this table, respectively. The final SQL command is then composed by following the syntax for the INSERT SQL command. Further, in the above example we retrieve the automatically generated keys. Therefore, we invoked the appropriate executeUpdate method and passed the Statement.RETURN_GENERATED_KEYS argument, which guarantees that the newly generated keys are returned. The generated keys might be then retrieved from the statement instance by invoking getGeneratedKeys() method. This method returns an instance of the ResultSet class. More on the ResultSet class, and how its objects are used to retrieve the database content in the next section. Finally, we need to catch SQLException again because this exception will be thrown if there an error occurs during the execution of the executeUpdate method.

The complete code of the Persistent class, implementing the above functionality can be found online at [http://coronet.iicm.edu/shop/src/shop/db/Persistent.java](http://coronet.iicm.edu/shop/src/shop/db/Persistent.java). The above code is from the store() method.

Retrieving Data from a Database
Similarly, to retrieve data from a database we need to obtain an instance of the Statement class, and then to invoke executeQuery() method on this instance. This method takes as an argument the SQL SELECT command representing the query.

**Example 7. Retrieving Data from a Database**

```java
... try{
    ResultSet query_result = statement.executeQuery(query);
} catch(SQLException exc) {
    exc.printStackTrace();
}
... 
```

Note, that the "query" argument should contain the appropriate SQL SELECT method. The executeQuery() method returns an instance of the ResultSet class.

The complete code of the Persistent class, implementing the above functionality can be found online at [http://coronet.iicm.edu/shop/src/shop/db/Persistent.java](http://coronet.iicm.edu/shop/src/shop/db/Persistent.java). The above code is from the getObjects() method.

**Retrieving Values from Result Sets**

Generally, execution of any JDBC statement that returns some data from a database results in an instance of the ResultSet class. This instance contains a number of rows (records) that hold the current results. The number and type of columns in this object corresponds to the number and types of columns returned as the result from the database system. For example, suppose we issue the following SELECT command to the "example" database.

**Example 8. Retrieving Data from a Database**

```sql
SELECT * FROM Customer;
```

This SELECT command retrieves all records from the Customer table. Recollect the definition of the Customer table.

**Relations**

<table>
<thead>
<tr>
<th>Relation Name</th>
<th>Relation Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>(cname, ccity, cphone, cid)</td>
</tr>
</tbody>
</table>

**Domains**
Thus, executing this SELECT command will result in obtaining an instance of the ResultSet class which will hold all records from the Customer table as rows. Further, each row will contain 4 values: cname, ccity, cphone, and cid with types String, String, Integer, Integer, respectively.

The next() Method

Basically, an instance of the ResultSet class is an iterator over the rows it keeps. There is always the current row, and at a specific moment we can only obtain data from the current row. If we want to move the cursor to the next row we need to invoke the next() method. At the beginning, the current row is set before the first row of the result, thus before we can read data from the first row we must invoke the next() method.

Example 9. Invoking the next() Method

```java
try{
    ResultSet query_result = statement.executeQuery(query);
    while(query_result.next()){
        ...
    }
}catch(SQLException exc){
    exc.printStackTrace();
}
```

In the above example we iterate through all the rows contained in the result.

The complete code of the Persistent class, implementing the above functionality can be found online at [http://coronet.iicm.edu/shop/src/shop/db/Persistent.java](http://coronet.iicm.edu/shop/src/shop/db/Persistent.java). The above code is from the getObjects() method.

The getXXX() Methods

Once when we have the current row of an instance of the ResultSet class, we may invoke different methods to retrieve the values of the columns. These methods and their names always correspond to the type of a particular column. Thus, to retrieve the value of a column with the type String we need to invoke the getString() method. Similarly, to retrieve an integer value we would call the getInt() method.
Example 10. Invoking the getXXX() Methods

... 
try{
    ResultSet query_result = statement.executeQuery(query);
    while(query_result.next()){
        String cname = query_result.getString("cname");
        ...
        int cid = query_result.getInt("cid");
        ...
    }
}catch(SQLException exc){
    exc.printStackTrace();
}
...

Note, that you need to take care of the appropriate types and the order in which you invoke the getXXX() methods. If, for instance, you invoke a method which expects a different column type or name a SQLException will be thrown.

Finally, you may acces the online application, which provides a Web interface to access and manipulate the contebt of the "example" database at http://coronet.iicm.edu/shop/.
Note, that this Web application is implemented by means of the Java Servlet technology.

Normally, the SQL statement explicitely defines the "ResultSet" internal structure. Consider the following sample database:

- Customer(cn,cname,ccity);
- Product(pn,pname,pprice);
- Transaction(cn,pn,tdate,tqnt);

The "executeQuery" may contain parameters received via an HTTP GET request.

... 
response.setContentType("text/html");
PrintWriter writer = response.getWriter();
String customerCity = request.getParameter("customerCity");
try { String sql = "SELECT cname, pname, qnt";
    sql = sql + " FROM Customer, Product, Transaction";
    sql = sql + " where Customer.ccity = \\
    sql = sql + " Customer.cn = Transaction.cn And";
    sql = sql + "Transaction.pn = Product.pn";
    ResultSet query_result = statement.executeQuery(sql);
    while(query_result.next()) { ... } ...

// please note that the example is written without taking into account
// a so-called
// "SQL Injection". ...

To comprehend the basic principles of "SQL Injection", suppose the user pass a string like this
as a value for the parameter customeId (cn). Obviously the string insert_sql will look as:

```
INSERT INTO Customer VALUES('1','1','1'); DELETE FROM Transaction;
```

```

response.setContentType("text/html");
PrintWriter writer = response.getWriter();
String cn = request.getParameter("customerID");
String customerName = request.getParameter("customerName");
String customerCity = request.getParameter("customerCity");
try
{
String insert_sql = "INSERT INTO Customer ";
insert_sql += "VALUES(" + cn + "," + customerName + "," + customerCity + ");"
statement = connection.createStatement();
statement.executeUpdate(insert_sql);
```

All the tuples of the relation "Transaction" will be deleted!

To prevent such SQL injection, so-called prepared statements can be used. In this case, SQL query is not defined as a string containing the source text. SQL query is a prepared statement with a number of parameters that can be set at run-time.

```
Consider the following sample database:
```

```
Customer(cn,cname,ccity);
Product(pn,pname,pprice);
Transaction(cn,pn,tdate,tqnt);
```

```
response.setContentType("text/html");
PrintWriter writer = response.getWriter();
String cn = request.getParameter("customerID");
String customerName = request.getParameter("customerName");
String customerCity = request.getParameter("customerCity");
try
{
String insert_sql = "INSERT INTO Customer " ;
insert_sql += "VALUES(?,?,?)"
PreparedStatement statement = connection.prepareStatement(insert_sql);
statement.setString(1, cn);
statement.setString(2, customerName);
statement.setString(3, customerCity);
statement.executeUpdate();
```

```
Once when we set a current row of the ResultSet, we can retrieve values by means of a number of methods. The methods correspond to a column type. Thus, to retrieve the value of a string column, we invoke a getString() method. Similarly, to retrieve an integer value we simply invoke a getInt() method.
```
response.setContentType("text/html");
PrintWriter writer = response.getWriter();
String customerCity = request.getParameter("customerCity");
try {
    String sql = "SELECT cname, pname, qnt";
    sql = sql + " FROM Customer, Product, Transaction";
    sql = sql + " where Customer.ccity = "+ customerCity + " And";
    sql = sql + " Customer.cn = Transaction.cn And";
    sql = sql + " Transaction.pn = Product.pn";
    ResultSet query_result = statement.executeQuery(sql);
    while(query_result.next()) {
        String customerName = query_result.getString("cname");
        String productTitle = query_result.getString("pname");
        int productQuantity = query_result.getInt("qnt"); ...
    }
    ...

Bibliography

Online Resources


