ABOUT YATIN BATRA

Yatin has graduated in Electronics & Telecommunication. During his studies, he has been involved with a large number of projects ranging from programming and software engineering to telecommunications analysis. He works as a software developer in the information technology sector where he is mainly involved with projects based on Java and J2EE technologies platform.

JDBC Best Practices Tutorial

Posted by: Yatin Batra in sql July 25th, 2017

Hello, in this tutorial we will learn some **Java Database Connectivity (JDBC) best practices** that Java programmer should follow while writing JDBC code. JDBC API is used to connect and interact with the relational databases to perform CREATE, READ, UPDATE, DELETE (commonly known as CRUD) operations. It is a database-independent API that you can use to execute your queries against a database. In this article, I will present the best practices that anyone should follow when using JDBC.

Table Of Contents

1. JDBC Components

The core JDBC components are comprised of the following:

- **JDBC Driver**: This is a collection of classes that enables you to connect to a database and perform CRUD operations against it.
- **Connection**: This class is used to connect to a database using the JDBC API. Developers can obtain a connection to a database only after the JDBC driver for that database is loaded and initialized in JVM memory.
- **Statement**: A statement is used to execute the CRUD operations.
- **Result Set**: After developers have executed a query using the JDBC API, the result of the query is returned in the form of a

```
ResultSet
```

.

The following is a list of the possible use cases in JDBC:

- Query Database
- Query Database Metadata
- Update Database

2. JDBC Best Practices

In this section, we will explore the strategies that can be adopted to improve JDBC performance.

2.1 Use Prepared Statement

It is very important JDBC best practice. **Prepared Statement** is used for executing a precompiled SQL statement.

```
java.sql.PreparedStatement
```

is suitable for executing DML commands: SELECT, INSERT, UPDATE and DELETE. Prepared Statement is faster as compared to Statement because it is used for executing pre-compiled SQL statements. Hence, same SQL query **can** be **executed repeatedly** in Prepared Statement.

Here is an example of how to use

```
PreparedStatement
```

in Java:

PreparedStmtExample.java

```
01 package com.jcg.jdbc.bestpractices;
02
03 import java.sql.Connection;
04 import java.sql.DriverManager;
05 import java.sql.PreparedStatement;
06 import java.sql.ResultSet;
07
08 public class PreparedStmtExample {
09
10     // JDBC Driver Name & Database URL
11     static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
12     static final String JDBC_DB_URL = "jdbc:mysql://localhost:3306/tutorialDb";
13
14     // JDBC Database Credentials
15     static final String JDBC_USER = "root";
16     static final String JDBC_PASS = "admin@123";
17
18     public static void main(String[] args) {
19         try {
20             Class.forName(JDBC_DRIVER);
21             Connection connObj = DriverManager.getConnection(JDBC_DB_URL, JDBC_USER, JDBC_PASS);
22
23             PreparedStatement prepStatement = connObj.prepareStatement("SELECT DISTINCT loan_type FROM bank
24             prepStatement.setString(1, "Citibank");
25
26             ResultSet resObj = prepStatement.executeQuery();
27             while(resObj.next()) {
28                 System.out.println("Loan Type?= " + resObj.getString("loan_type"));
29             }
30         } catch (Exception sqlException) {
31             sqlException.printStackTrace();
32         }
33     }
34 }
```

2.2 Use Connection Pool

It is a very common JDBC best practice to use Connection pooling in Java. **Connection pooling** is the process where we maintain a cache of database connections. Database connections maintained in a cache can be reused whenever a request comes to connect with the database. So, Connection pooling reduces database hits and improves the application performance significantly.

Application servers allow configuration of JDBC connection pools where developers can define the minimum and a maximum number of database connections that could be created within the application server. The application server manages the creation and deletion of database connections. JNDI Lookup is used in the application to obtain the database connections from the pool.

There are a few choices when using the JDBC connection pool:

- Developers can depend on application server if it supports this feature, generally, all the application servers support connection pools. Application server creates the connection pool on behalf of developers when it starts. Developers need to give properties like min, max and incremental sizes to the application server.
- Developers can use JDBC 2.0 interfaces, for e.g.

  ```
  ConnectionPoolDataSource
  ```

  and

  ```
  PooledConnection
  ```

  if the driver implements these interfaces.
- Developers can even create their own connection pool if they are not using any application server or JDBC 2.0 compatible driver.

By using any of these options, one can increase the JDBC performance significantly.

mode disable. The reason behind this JDBC best practice is that with auto commit mode disabled we can group SQL Statement in one transaction, while in the case of auto commit mode enabled every SQL statement runs on its own transaction and committed as soon as it finishes. So, always **execute SQL queries with auto commit mode disabled**.

- Developers can set auto commit mode of connection to false using

  ```
  connObj.setAutoCommit(false)
  ```

  and then accordingly use

  ```
  connObj.commit()
  ```

  or

  ```
  connObj.rollback()
  ```

  .
- If any transaction fails in between, then rollback the transaction by calling

  ```
  connObj.rollback()
  ```

  , and commit the transaction by using

  ```
  connObj.commit()
  ```

  only if it went successfully.

For e.g.:

Let's say we have to update salary of two employees, and salary of both employees must be updated simultaneously in a database. And let's say the salary of the first employee is updated successfully. But, if anything goes wrong in updating salary of the second employee then any modifications done to first employee's salary will be rolled back.

The following example illustrates the use of a commit and rollback object:

AutoCommitExample.java

```
01 package com.jcg.jdbc.bestpractices;
02
03 import java.sql.Connection;
04 import java.sql.DriverManager;
05 import java.sql.Statement;
06
07 public class AutoCommitExample {
08
09     // JDBC Driver Name & Database URL
10     static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
11     static final String JDBC_DB_URL = "jdbc:mysql://localhost:3306/tutorialDb";
12
13     // JDBC Database Credentials
14     static final String JDBC_USER = "root";
15     static final String JDBC_PASS = "admin@123";
16
17     public static void main(String[] args) {
18         try {
19             Class.forName(JDBC_DRIVER);
20             Connection connObj = DriverManager.getConnection(JDBC_DB_URL, JDBC_USER, JDBC_PASS);
21
22             // Assuming A Valid Connection Object
23             connObj.setAutoCommit(false);
24             Statement stmtObj = connObj.createStatement();
25
26             String correctQuery = "INSERT INTO employee VALUES (001, 20, 'Java', 'Geek')";
27             stmtObj.executeUpdate(correctQuery);
28
29             // Submitting A Malformed SQL Statement That Breaks
30             String incorrectQuery = "INSERTED IN employee VALUES (002, 22, 'Harry', 'Potter')";
31             stmtObj.executeUpdate(incorrectQuery);
32
33             // If There Is No Error.
34             connObj.commit();
35
36             // If There Is Error
37             connObj.rollback();
38         } catch (Exception sqlException) {
39             sqlException.printStackTrace();
40         }
41     }
42 }
```

2.4 Use JDBC Batch Statements

This is another JDBC best practice which is very popular among developers. JDBC API provides

```
addBatch()
```

method to add SQL queries into a batch and

```
executeBatch()
```

In simple words, Batch statement sends multiple requests from Java to the database in just one call. Without batch statements multiple requests will be sent in multiple (one by one) calls to the database.

About

`addBatch()`

method:

- `PreparedStatement`

  extends

  `Statement`

  and inherits all methods from

  `Statement`

  and additionally adds

  `addBatch()`

  method.

- `addBatch()`

  method adds a set of parameters to the

  `PreparedStatement`

  object's batch of commands.

The following example illustrates the use of batch statements:

BatchStatementsExample.java

```
01 package com.jcg.jdbc.bestpractices;
02
03 import java.sql.Connection;
04 import java.sql.DriverManager;
05 import java.sql.Statement;
06
07 public class BatchStatementsExample {
08
09     // JDBC Driver Name & Database URL
10     static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
11     static final String JDBC_DB_URL = "jdbc:mysql://localhost:3306/tutorialDb";
12
13     // JDBC Database Credentials
14     static final String JDBC_USER = "root";
15     static final String JDBC_PASS = "admin@123";
16
17     public static void main(String[] args) {
18         try {
19             Class.forName(JDBC_DRIVER);
20             Connection connObj = DriverManager.getConnection(JDBC_DB_URL, JDBC_USER, JDBC_PASS);
21
22             connObj.setAutoCommit(false);
23
24             Statement stmtObj = connObj.createStatement();
25             stmtObj.addBatch("INSERT INTO student VALUES(101, 'JavaGeek', 20)");
26             stmtObj.addBatch("INSERT INTO student VALUES(102, 'Lucifer', 19)");
27             stmtObj.addBatch("UPDATE employee SET age = 05 WHERE id = 001");
28
29             // Execute Batch
30             int[] recordsAffected = stmtObj.executeBatch();
31             connObj.commit();
32         } catch (Exception sqlException) {
33             sqlException.printStackTrace();
34         }
35     }
36 }
```

2.5 Accessing Result Set by Column Names

JDBC API allows accessing the returned data by SELECT query using

`ResultSet`

, which can further be accessed using either column name or the column index. This JDBC best practice suggests using *column name* over column index in order to avoid

`InvalidColumnIndexException`

column index starts from 1 and 0 is invalid.

Some Java programmers may argue that accessing a database column using index is faster than a name, which is true. But if we look in terms of maintenance, robustness, and readability, I prefer accessing the database column using the name in

```
ResultSet
```

iterator.

The following example illustrates the use:

InvalidColumnIndexExample.java

```
01 package com.jcg.jdbc.bestpractices;
02
03 import java.sql.Connection;
04 import java.sql.DriverManager;
05 import java.sql.PreparedStatement;
06 import java.sql.ResultSet;
07
08 public class InvalidColumnIndexExample {
09
10     // JDBC Driver Name & Database URL
11     static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
12     static final String JDBC_DB_URL = "jdbc:mysql://localhost:3306/tutorialDb";
13
14     // JDBC Database Credentials
15     static final String JDBC_USER = "root";
16     static final String JDBC_PASS = "admin@123";
17
18     public static void main(String[] args) {
19         try {
20             Class.forName(JDBC_DRIVER);
21             Connection connObj = DriverManager.getConnection(JDBC_DB_URL, JDBC_USER, JDBC_PASS);
22
23             PreparedStatement prepStmtObj = connObj.prepareStatement("SELECT DISTINCT item FROM order where
24             prepStmtObj.setString(0, "101"); // This Will Throw "java.sql.SQLException: Invalid Column Inde
25
26             ResultSet resultSetObj = prepStmtObj.executeQuery();
27             while(resultSetObj.next()) {
28                 System.out.println("Item: " + resultSetObj.getString(2)); // This Will Throw "java.sql.SQLE
29             }
30         } catch (Exception sqlException) {
31             sqlException.printStackTrace();
32         }
33     }
34 }
```

2.6 Use Bind variables instead of String Concatenation

In JDBC best practices, we have suggested using

```
PreparedStatement
```

in Java because of better performance. But performance can only be improved if developer use

```
bind variables
```

denoted by

```
?
```

or

```
place holders
```

which allow the database to run the same query with a different parameter. This JDBC best practices results in a better performance and provides protection against SQL injection as a text for all the parameter values is escaped.

Sample Code 1

```
1 prepStmt = con.prepareStatement("select * from EMPLOYEE where ID=? ");
2 prepStmt.setInt(1, 8);
```

While Statement **enforces SQL injection** because we end up using query formed using concatenated SQL strings.

Sample Code 2

```
1 String query = "select * from EMPLOYEE where id = ";
2 int i = 2 ;
3 stmt.executeQuery(query + String.valueOf(i));
```

Here comes one very important question, are **Prepared Statement vulnerable to SQL injections**? The answer is **yes** when we use concatenated SQL strings rather than using input as a parameter for the prepared statement.

It's a common Java coding practice to close any resource in

```
finally
```

block as soon as we are done using the resource. JDBC connection and classes are a costly resource and should be closed in

```
finally
```

block to ensure the release of connection even in the case of any

```
SQLException
```

. This even helps to avoid

```
ora-01000 - java.sql.SQLException
```

errors in Java.

The following example illustrates the use:

CloseJdbcObjects.java

```
01 package com.jcg.jdbc.bestpractices;
02
03 import java.sql.Connection;
04 import java.sql.PreparedStatement;
05 import java.sql.ResultSet;
06 import java.sql.SQLException;
07
08 public class CloseJdbcObjects {
09
10     public static void main(String[] args) throws ClassNotFoundException, SQLException {
11         Connection connObj = null;
12         PreparedStatement prepStmtObj = null;
13         ResultSet resultSetObj = null;
14         try {
15             // Business Logic!
16         }
17         finally{
18             try {
19                 // Close Result Set Object
20                 if(resultSetObj!=null) {
21                     resultSetObj.close();
22                 }
23                 // Close Prepared Statement Object
24                 if(prepStmtObj!=null) {
25                     prepStmtObj.close();
26                 }
27                 // Close Connection Object
28                 if(connObj!=null) {
29                     connObj.close();
30                 }
31             } catch (SQLException sqlException) {
32                 sqlException.printStackTrace();
33             }
34         }
35     }
36 }
```

From Java 7 onwards, developers can use the Automatic Resource Management (ARM) block to close resources automatically.

2.8 Statement Caching

Statement caching improves performance by caching executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. Starting from JDBC 3.0, JDBC API defines a statement-caching interface. Setting the

```
MaxPooledStatements
```

connection option enables the **statement pooling**. Enabling statement pooling allows the JDBC driver to re-use Prepared Statement objects and are returned to the pool instead of being freed and the next Prepared Statement with the same SQL statement is retrieved from the pool rather than being instantiated and prepared against the server.

Statement caching can:

- Prevent the overhead of repeated cursor creation.
- Prevent repeated statement parsing and creation.
- Allows the JDBC driver to use the Prepared Statement objects.

The following code snippet illustrates how Statement pooling can be enabled:

Sample Code 1

```
1 Properties propObj = new Properties();
2 propObj.setProperty("user", "root");
3 propObj.setProperty("password", "admin@123");
4 propObj.setProperty("MaxPooledStatements", "250");
```

2.9 Use correct getXXX() method

`ResultSet`

interface provides a lot of

`getXXX()`

methods to get and convert database data types to Java data types and is flexible in converting non-feasible data types. For e.g.:

- `getString(String columnName)`

  returns Java String object.
- Column Name is recommended to be a `VARCHAR`

  or `CHAR`

  type of database but it can also be a `NUMERIC`

  , `DATE`

  etc.

If you give not recommended parameters, it needs to cast it to proper Java data type that is expensive. For e.g., Consider that we want to select a product's id from a huge database which returns millions of records from search functionality. It needs to convert all these records that are very expensive.

So always use proper

`getXXX()`

methods according to JDBC recommendations.

2.10 Use standard SQL Statements

This is another JDBC best practice in Java which ensures writing portable code. Since most of JDBC code is filled up with SQL query it's easy to start using database specific feature which may present in MySQL but not in Oracle etc. By using ANSI SQL or by not using DB specific SQL we ensure minimal change in the DAO layer in case developers switch to another database.

Also, it is a good JDBC practice to write as much business logic as much as possible in **Stored Procedure** or **Functions** as compared to writing it down in Java class. Because this approach reduces the database hits and improves application performance significantly.

2.11 Choose suitable JDBC Driver

There are 4 types of JDBC drivers in Java and it can directly affect the performance of DAO layer. It is recommended to always use latest JDBC drivers if available and prefer type 4 native JDBC Drivers.

That's all for this post. Happy Learning!!

3. Conclusion

The main goal of this article is to discuss important and best JDBC (Java Database Connectivity) practices in Java with examples.

4. Download the Eclipse Project

This was an example of JDBC Best Practices.

Download
You can download the full source code of this example here: **JDBC Best Practices**

Tagged with: CORE JAVA JDBC

KNOWLEDGE BASE

Courses
News
Resources
Tutorials
Whitepapers

THE CODE GEEKS NETWORK

.NET Code Geeks
Java Code Geeks
System Code Geeks
Web Code Geeks

HALL OF FAME

Android Alert Dialog Example
Android OnClickListener Example
How to convert Character to String and a String to Character Array in Java
Java Inheritance example
Java write to File Example
java.io.FileNotFoundException – How to solve File Not Found Exception
java.lang.arrayindexoutofboundsexception – How to handle Array Index Out Of Bounds Exception
java.lang.NoClassDefFoundError – How to solve No Class Def Found Error
JSON Example With Jersey + Jackson
Spring JdbcTemplate Example

ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community focused on crea ultimate Java to Java developers resource center; targeted at the technical a technical team lead (senior developer), project manager and junior developer JCGs serve the Java, SOA, Agile and Telecom communities with daily news w domain experts, articles, tutorials, reviews, announcements, code snippets ar source projects.

DISCLAIMER