



SOEN 387

WEB-BASED ENTERPRISE APPLICATIONS DESIGN

TUTORIAL – 5 Using Databases

By
Vasu Ratanpara



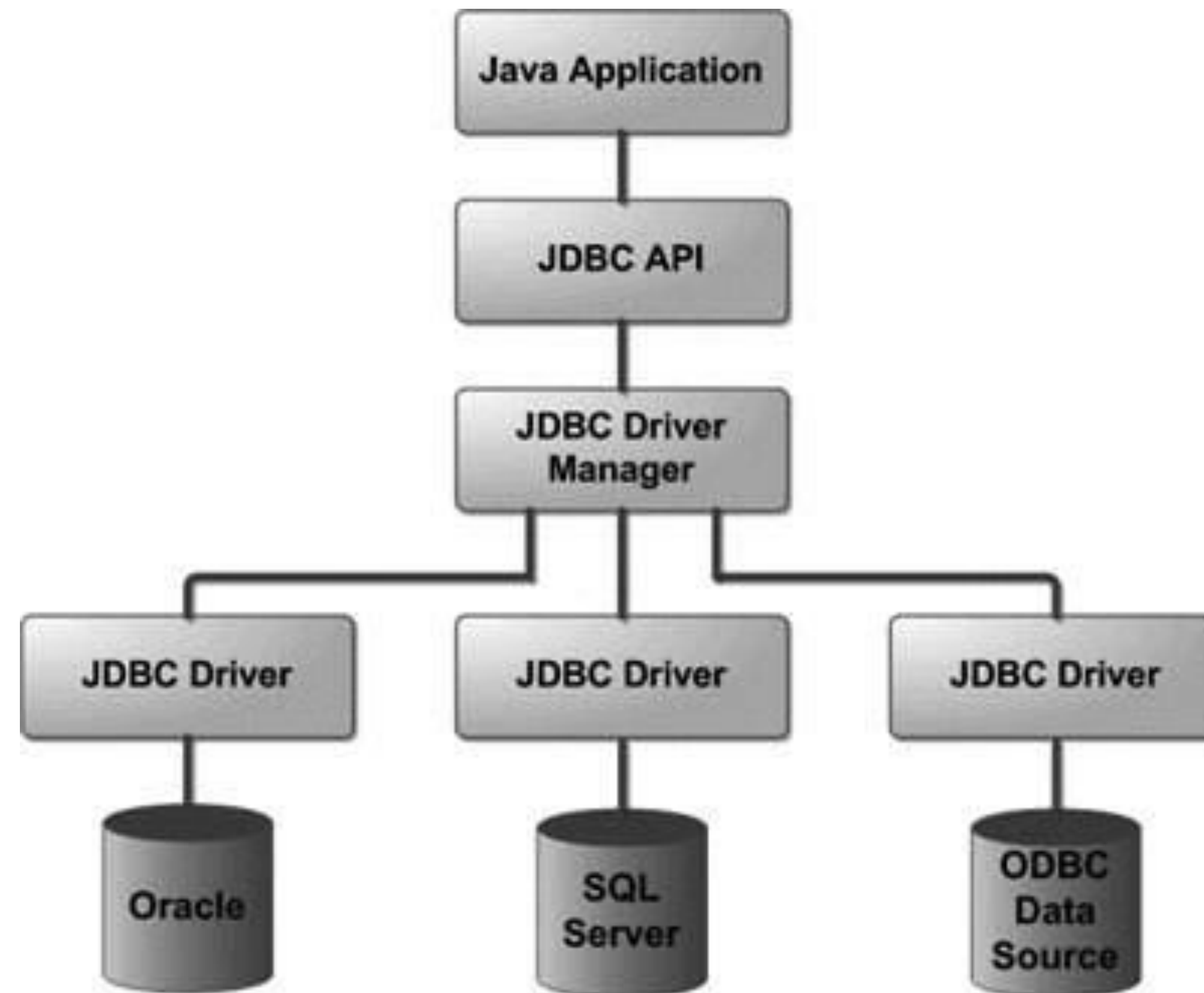
What is JDBC?

JDBC

- ✓ JDBC stands for **java database connectivity**, which is a standard java API for database independent connectivity between the java programming language and a wide range of databases.
- ✓ JDBC provides Java programs to contain database-independent code.
- ✓ The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.
 - Making a connection to a database.
 - Creating SQL or MySQL statements.
 - Executing SQL or MySQL queries in the database.
 - Viewing & modifying the resulting records.



JDBC Architecture

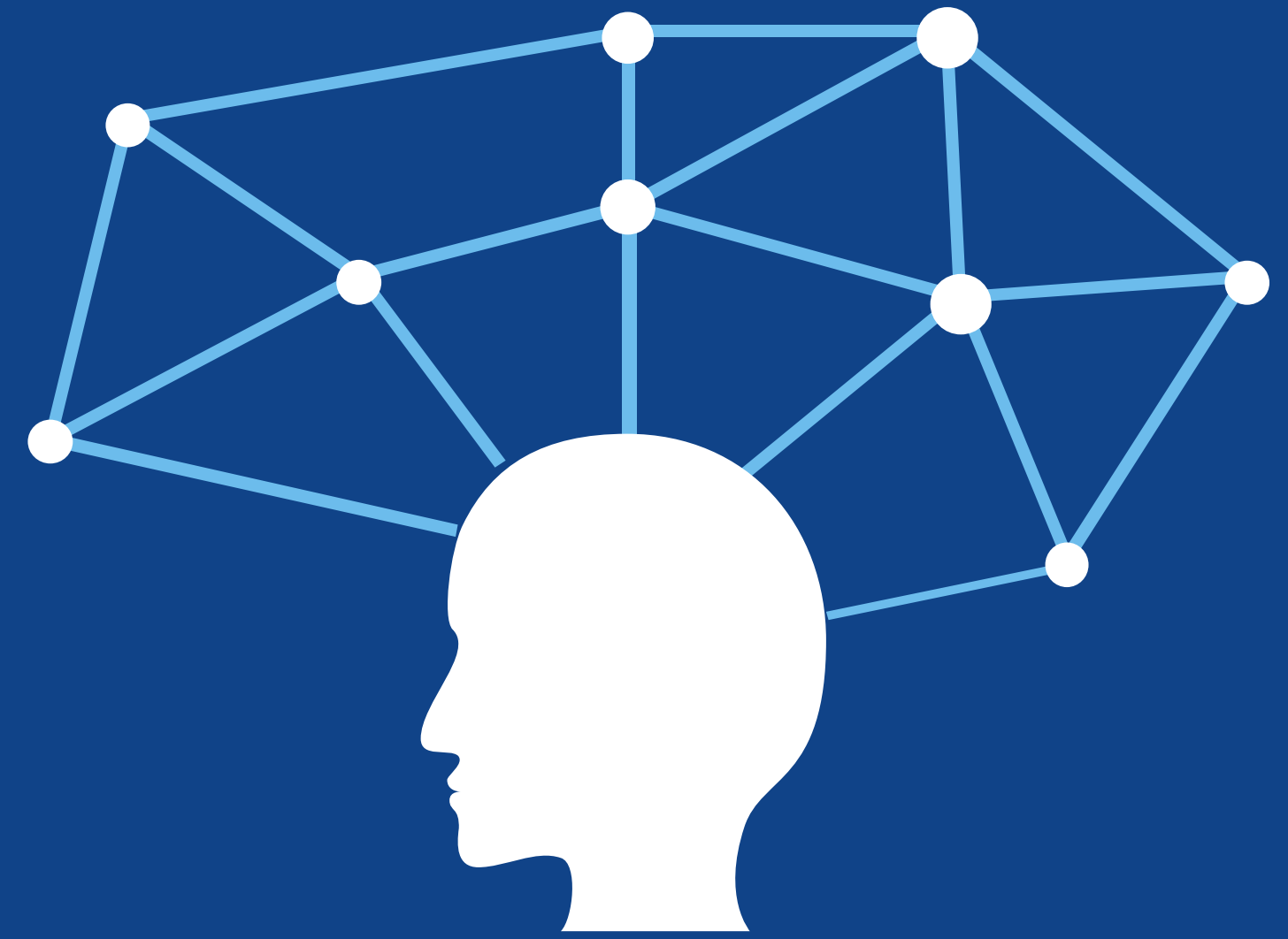


JDBC : Common Components

- ✓ DriverManager
- ✓ Driver
- ✓ Connection
- ✓ Statement
- ✓ ResultSet
- ✓ SQLException

Enough Talking

Let's start coding



Create Database

- ✓ Let's create Database of our university in MySQL, which can store student's details.
- ✓ To create Database and use it follow given syntax:

Syntax: CREATE DATABASE DATABASE_NAME;

SQL> CREATE DATABASE Concordia;

Syntax: USE DATABASE_NAME;

SQL> USE Concordia;

- ✓ Create a Table name user_details in Concordia database;

Syntax: CREATE TABLE table_name (`column1` DATATYPE , `column2` DATATYPE, ...) ;

SQL> CREATE TABLE `user_details` (`user_id` int(11) NOT NULL, `username` varchar(255) DEFAULT NULL, `first_name` varchar(50) DEFAULT NULL, `last_name` varchar(50) DEFAULT NULL, `gender` varchar(10) DEFAULT NULL, `password` varchar(50) DEFAULT NULL, `status` tinyint(10) DEFAULT NULL);

Insert data into Database

- ✓ Create a Table name user_details in Concordia database;

Syntax: INSERT INTO table_name VALUES (column1, column2, ...);

SQL> INSERT INTO `user_details` (`user_id`, `username`, `first_name`, `last_name`, `gender`,
`password`, `status`) VALUES (1, 'rogers63', 'david', 'john', 'Female',
'e6a33eee180b07e563d74fee8c2c66b8', 1);

- ✓ NOTE: Add more data in your database from [here](#)



JDBC: Basic steps to use a database in Java

1. Establish a connection
2. Create JDBC Statements
3. Execute SQL Statements
4. GET ResultSet (IF any)
5. Close connections



Step1: Establish a connection

```
// JDBC driver name and database URL
```

```
String JDBC_DRIVER =
```

```
"com.mysql.cj.jdbc.Driver";
```

```
String DB_URL =
```

```
"jdbc:mysql://localhost:3306/";
```

```
String DB_NAME = "concordia";
```

```
// Database credentials
```

```
String DB_USER = "root";
```

```
String DB_PASSWORD = "";
```

NOTES:

1. If you are using any other database like PostgreSQL etc. you have to change your JDBC driver as well as DB URL accordingly. Also, All the above code is in Main() method.

2. Here use your own DB username and DB password

3. Class `com.mysql.jdbc.Driver` This is deprecated. The new driver class is `com.mysql.cj.jdbc.Driver`.

(It will work with old class but with warning)

Step1: Establish a connection (Cont.)

....

try

{

// Register JDBC driver

Class.forName(JDBC_DRIVER);

//Open a connection

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

Connection conn =

DriverManager.getConnection(DB_URL+DB_NAME,DB_USER,DB_PASSWORD);


....

}




Step2: Create JDBC Statements

```
try
{
    ....
    //Create JDBC Statements
    System.out.println("Creating statement...");
    stmt = conn.createStatement();
    ....
}
```



Step3: Execute SQL Statements

```
try
{
    ....
    // Execute a query
    String sql = "SELECT user_id, first_name, last_name FROM user_details";
    ResultSet rs = stmt.executeQuery(sql); // We may not have ResultSet for other queries
    ....
}
```




Step4: GET ResultSet (If Applicable)

```
try
{
    ....


    // Extract data from result set
    while(rs.next())
    {
        //Retrieve by column name
        int id  = rs.getInt("user_id");
        String firstName = rs.getString("first_name");
        String lastName = rs.getString("last_name");
        // User your data here (i.e print or store in ArrayList)

    }
    ....
}
```



Step5: Close connections

```
try
{
    //Close connections to clean-up environment
    rs.close();
    stmt.close();
    conn.close();
}
// Add more catch statements (i.e. for SQLException)
catch(Exception e){
    // handle your exceptions here
}
```



DRY Rule

- ✓ Don't repeat yourself
- ✓ Reduce redundant code
- ✓ Less LOC (Line of Code)

Now Consider a situation when you need to run more than 100+ SQL queries. Imagine yourself writing the above steps again and again.

To avoid this we will create a new class which will handle the work of DB Connection and we will use a static method (think why!) which will return a DB Connection Object and later on we can reuse it as many time as we want.

(We will call that file as DBConnection.java)



DBConnection.Java

```
import java.sql.*;

public class DBConnection
{
    // NOTE : It'd be better to get them from a properties or XML configuration file
    // JDBC driver name and database URL
    static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost:3306/";
    static final String DB_NAME = "concordia";
    // Database credentials
    static final String DB_USER = "root";
    static final String DB_PASSWORD = "";
    static Connection conn = null;
```

.....

DBConnection.Java (Cont.)

.....

```
public static Connection getConnection() {  
    try{  
        Class.forName(JDBC_DRIVER); //Register JDBC driver  
        //Open a connection  
        conn = DriverManager.getConnection(DB_URL+DB_NAME,DB_USER,DB_PASSWORD);  
        return conn;  
    } catch (SQLException e) {  
        throw new RuntimeException("Error connecting to database",e);  
    } catch (ClassNotFoundException e){  
        throw new RuntimeException("Error Class Not Found",e);  
    }  
}
```

.....

DBConnection.Java (Cont.)

.....

```
public static void closeConnection() throws SQLException{  
    //Close connection  
    if(conn!=null) conn.close();  
}  
}
```



Insert Record using JDBC

```
try
{
    ....

    stmt = conn.createStatement();
    // insert execute query
    String query = "INSERT INTO user_details VALUES (1, 'rogers63', 'david', 'john', 'Male', '123', 1)";
    stmt.executeUpdate(query);

    String query = "INSERT INTO user_details VALUES (2, 'mike28', 'rogers', 'paul', 'Male', '345', 0)";
    stmt.executeUpdate(query);


    ....
}
```



Select Record using JDBC


```
try
{
    ....
    // select execute query
    String sql = "SELECT user_id, username FROM user_details";
    ResultSet rs = stmt.executeQuery(sql);
    // Extract data from result set
    while(rs.next()) {
        //Retrieve by column name
        int id = rs.getInt("user_id");
        String first = rs.getString("username");
        // User your data here (i.e. print or store in Array list)

    }
    ....
}
```




Update Record using JDBC

```
try
{
    ....
    // Update execute query
    String sql = "UPDATE user_details SET status = 0 WHERE id = 5";
    stmt.executeUpdate(sql);
    // Now you can extract all the records to see updated records
    // Hint: Use Select query
    ....
}
```



Delete Record using JDBC

```
try
{
    ....
    // Delete execute query
    String sql = "DELETE FROM user_details" + " WHERE id = 5";
    stmt.executeUpdate(sql);
    // Now you can extract all the records to see remaining records
    ....
}
```



DAO Pattern

- ✓ The Data Access Object (DAO) pattern is a structural pattern that allows us to isolate the application/business layer from the persistence layer (usually a relational database, but it could be any other persistence mechanism) using an abstract API.
- ✓ The functionality of this API is to hide from the application all the complexities involved in performing CRUD operations in the underlying storage mechanism. This permits both layers to evolve separately without knowing anything about each other.



“

Let's see an Example of DAO pattern



Using Parameters

- ✓ There are three types of parameters which exist :
- ✓ Which are IN, OUT, and INOUT.
- ✓ The PreparedStatement object only uses the IN parameter.
- ✓ The CallableStatement object can use all the three.

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

Creating a procedure

```
DELIMITER $$
```

```
DROP PROCEDURE IF EXISTS 'EMP' . 'getEmpName' $$
```

```
CREATE PROCEDURE 'EMP' . 'getEmpName' ( IN EMP_ID INT , OUT EMP_FIRST  
VARCHAR(255))
```

```
BEGIN
```

```
    SELECT first INTO EMP_FIRST
```

```
    FROM Employees
```

```
    where ID= EMP_ID;
```

```
END $$
```

```
DELIMITER;
```



CallableStatement

```
CallableStatement cstmt = null;
```

```
try {
```

```
    String SQL = "{call getEmpName (?, ?)}";
```

```
    cstmt = conn.prepareCall (SQL);
```

```
    ...
```

```
} catch (SQLException e) {
```

```
    ...
```

```
} finally {
```


```
    cstmt.close();
```

```
}
```



Transactions in JDBC using Commit & Rollback

```
try{  
    ....  
    conn.setAutoCommit(false);  
    Statement stmt = conn.createStatement();  
    String SQL = "INSERT INTO user_details VALUES (3, 'rivera92', 'david', 'john', 'Male', 'pwd', 1)";  
    stmt.executeUpdate(SQL);  
    //Submit a malformed SQL statement that breaks  
    String SQL = "INSERTED IN user_details VALUES (4, 'ross95', 'maria', 'sanders', 'Female', 123, 1)";  
    stmt.executeUpdate(SQL);  
    // If there is no error.  
    conn.commit();  
} catch(SQLException se){  
    // If there is any error.  
    conn.rollback();  
}
```



SQL Injection

```
// SQL query vulnerable  
String username = req.getParameter("username");  
String password = req.getParameter("password");  
String id = null;  
String sql = "SELECT user_id FROM user_deatils WHERE  
            username='"+username+"' AND password='"+password";  
stmt.executeQuery(sql);
```



SQL Injection(cont.)

- ✓ These input fields are vulnerable to SQL Injection.
- ✓ An attacker could use SQL commands in the input in a way that would alter the SQL statement executed by the database server.
- ✓ For example, they could use a trick involving a single quote and set the passwd field to:

Password' OR 1 = 1

- ✓ As a result, the database server runs the following SQL query:

SELECT id FROM users WHERE username='username' AND password='password' OR 1=1'




SQL Injection(cont.)

- ✓ Because of the OR 1=1 statement, the WHERE clause returns the first id from the users table no matter what the username and password are.
- ✓ The first user id in a database is very often the administrator. In this way, the attacker not only bypasses authentication but also gains administrator privileges.
- ✓ They can also comment out the rest of the SQL statement to control the execution of the SQL query further:

```
SQLite
' OR '1'='1' --
' OR '1'='1' /*
-- MySQL
' OR '1'='1' #
-- Access (using null characters)
' OR '1'='1' %00
' OR '1'='1' %16
```


Defence Against an SQL Injection Attack using PreparedStatement

- ✓ One of the ways to reduce the chance of SQL injection attack is to ensure that the unfiltered strings of text must not be allowed to appended to the SQL statement before execution.
 - ✓ For example, we can use **PreparedStatement** to perform required database tasks.
 - ✓ The interesting aspect of PreparedStatement is that it sends a **pre-compiled** SQL statement to the database, rather than a string.
 - ✓ This means that query and data are separately send to the database.
 - ✓ This **prevents** the root cause of the **SQL injection attack**, because in SQL injection, the idea is to mix code and data wherein the data is actually a part of the code in the guise of data.
- 

PreparedStatement(cont.)

- ✓ In PreparedStatement, there are multiple setXYZ() methods, such as setString().
- ✓ These methods are used to filter special characters such as a quotation contained within the SQL statements.
- ✓ PreparedStatement seems to be the better option between the two.
- ✓ It prevents malicious strings from being concatenated due to its different approach in sending the statement to the database.
- ✓ PreparedStatement uses variable substitution rather than concatenation.
- ✓ Placing a question mark (?) in the SQL query signifies that a substitute variable will take its place and supply the value when the query is executed.
- ✓ The position of the substitution variable takes its place according to the assigned parameter index position in the setXYZ() methods.
- ✓ This **technique** prevents it from **SQL injection attack**.

PreparedStatement(cont.)

```
try (PreparedStatement ps = conn.prepareStatement(selectQuery))
{
    ps.setInt(1,99);
    ResultSet rs = pstmt.executeQuery();
    ....
} catch (Exception ex){
    ex.printStackTrace();
}
```



References

- ✓ <https://dev.mysql.com/downloads/mysql/>
- ✓ <https://overiq.com/installing-mysql-windows-linux-and-mac/>
- ✓ <https://dev.mysql.com/doc/connectors/en/connector-j-installing-maven.html>
- ✓ <https://www.javatpoint.com/mysql-tutorial>
- ✓ <https://www.tutorialspoint.com/jdbc/index.htm>
- ✓ <https://dzone.com/articles/building-simple-data-access-layer-using-jdbc>
- ✓ <https://www.baeldung.com/java-dao-pattern>
- ✓ <https://www.acunetix.com/websecurity/sql-injection/>
- ✓ <https://www.developer.com/db/how-to-protect-a-jdbc-application-against-sql-injection.html>



“

Thank you

Any Questions?

