Lab 32

Database Connectivity in Java

Task: Write a report on Database Connectivity with sample codes

Certainly! Below is a report that includes Java code examples for connecting to a MySQL database using JDBC (Java Database Connectivity). The report provides an overview of the steps involved in database connectivity and includes code samples for each step.

Report on Database Connectivity in Java using JDBC

Introduction

Database connectivity is a crucial aspect of many software applications, enabling them to store and retrieve data from relational databases. In Java, this connectivity is achieved through the use of the Java Database Connectivity (JDBC) API. JDBC provides a standardized way to interact with various database management systems (DBMS) by using a set of Java classes and interfaces.

In this report, we will walk through the steps required to establish a database connection, execute SQL queries, and retrieve data from a MySQL database using JDBC. We will also provide Java code examples for each step.

Steps for Database Connectivity

1. Import JDBC Libraries

Before using JDBC, we need to import the necessary libraries. The JDBC libraries are typically included with your Java development environment or can be downloaded separately. The following import statement is used to include JDBC classes:

import java.sql.\*;

2. Load the JDBC Driver

Each DBMS requires a specific JDBC driver to be loaded. For MySQL, the driver class is `"com.mysql.jdbc.Driver"`. We use the `Class.forName()` method to load the driver class dynamically:

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

3. Establish a Database Connection

To connect to the database, we use the `DriverManager.getConnection()` method, passing the database URL, username, and password as parameters:

String url = "jdbc:mysql://localhost:3306/mydb"; // Replace with your database URL

String username = "username";

String password = "password";

Connection connection = DriverManager.getConnection(url, username, password);

4. Create a Statement or PreparedStatement

To execute SQL queries, we can create either a `Statement` or a `PreparedStatement` object. `PreparedStatement` is preferred for parameterized queries to prevent SQL injection:

Statement statement = connection.createStatement();

// or

PreparedStatement preparedStatement = connection.prepareStatement("SELECT \* FROM my\_table WHERE id = ?");

5. Execute SQL Queries

We can execute SQL queries using the `executeQuery()` method for SELECT queries and `executeUpdate()` for INSERT, UPDATE, DELETE, and Data Definition Language (DDL) queries:

ResultSet resultSet = statement.executeQuery("SELECT \* FROM my\_table");

// or

int rowsAffected = preparedStatement.executeUpdate();

6. Process the ResultSet

If a SELECT query was executed, we can process the results using a `ResultSet`. We iterate through the result set to retrieve data:

while (resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

// Process the data

}

7. Close Resources

To release system resources, it is essential to close the database resources when done with them:

resultSet.close();

statement.close();

connection.close();

8. Handle Exceptions

Database operations can throw exceptions, so it is crucial to handle them using try-catch blocks or propagate them to the calling method.

Code Examples

Here are simplified code examples that demonstrate database connectivity in Java using JDBC:

import java.sql.\*;

public class DatabaseConnectivityDemo {

public static void main(String[] args) {

try {

// Load the JDBC driver

Class.forName("com.mysql.cj.jdbc.Driver");

// Establish a database connection

String url = "jdbc:mysql://localhost:3306/mydb";

String username = "username";

String password = "password";

Connection connection = DriverManager.getConnection(url, username, password);

// Create a Statement

Statement statement = connection.createStatement();

// Execute a SELECT query

ResultSet resultSet = statement.executeQuery("SELECT \* FROM my\_table");

// Process the ResultSet

while (resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

System.out.println("ID: " + id + ", Name: " + name);

}

// Close resources

resultSet.close();

statement.close();

connection.close();

} catch (ClassNotFoundException | SQLException e) {

e.printStackTrace();

}

}

}

Conclusion

Database connectivity in Java using JDBC is a fundamental skill for building applications that interact with relational databases. By following the steps outlined in this report and using the provided code examples, developers can establish database connections, execute SQL queries, and retrieve data seamlessly. JDBC provides a powerful and standardized interface for working with a wide range of database management systems, making it an essential tool for Java developers.