Guía Avanzada de Conectividad de Bases de Datos con JDBC

1. Introducción a JDBC

Java Database Connectivity (JDBC) es una API que permite a Java interactuar con bases de datos relacionales de forma independiente de la base de datos específica. JDBC proporciona interfaces y clases que permiten realizar operaciones como conexión, consulta, actualización y cierre de conexiones de manera eficiente.

---

2. Arquitectura de JDBC

La arquitectura de JDBC incluye varios componentes que facilitan la conexión y manipulación de datos en una base de datos:

- **Driver Manager**: Administra un conjunto de controladores JDBC, facilitando la conexión con diversas bases de datos.
- **Driver**: Proporciona la implementación para interactuar con bases de datos específicas.
- **Connection**: Representa una conexión establecida con la base de datos.
- **Statement**: Permite ejecutar consultas SQL en la base de datos.
- **ResultSet**: Contiene los datos recuperados de la base de datos tras ejecutar una consulta.
- **SQLException**: Maneja las excepciones de base de datos.

---

3. Conexión a la Base de Datos con JDBC

Para conectarse a una base de datos, se sigue una serie de pasos que suelen involucrar:

1. **Registrar el controlador JDBC**
2. **Establecer una conexión**
3. **Crear un objeto Statement**
4. **Ejecutar consultas SQL**
5. **Cerrar la conexión**

Ejemplo básico de conexión

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnectionExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/nombreBaseDatos";
        String user = "usuario";
        String password = "contraseña";

        try (Connection connection =
DriverManager.getConnection(url, user, password)) {
            System.out.println("Conexión exitosa a la base de
datos");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

---

4. Ejecución de Consultas con JDBC

4.1 Tipos de Sentencias

- **Statement**: Para consultas SQL estáticas.
- **PreparedStatement**: Para consultas SQL parametrizadas, ayuda a prevenir SQL Injection y es eficiente para ejecuciones repetidas.
- **CallableStatement**: Ejecuta procedimientos almacenados en la base de datos.

4.2 Uso de Statement

```
String query = "SELECT * FROM usuarios";
try (Statement stmt = connection.createStatement();
     ResultSet rs = stmt.executeQuery(query)) {
    while (rs.next()) {
        System.out.println("Usuario: " + rs.getString("nombre"));
    }
}
```

4.3 Uso de PreparedStatement

```
String query = "SELECT * FROM usuarios WHERE edad > ?";
try (PreparedStatement pstmt = connection.prepareStatement(query)) {
    pstmt.setInt(1, 18);
    ResultSet rs = pstmt.executeQuery();
    while (rs.next()) {
        System.out.println("Usuario adulto: " +
rs.getString("nombre"));
    }
}
```

---

5. Manejo de Resultados con ResultSet

- **Tipos de ResultSet**:
  - **ResultSet.TYPE_FORWARD_ONLY**: Sólo avanza hacia adelante.
  - **ResultSet.TYPE_SCROLL_INSENSITIVE**: Permite moverse en ambas direcciones, sin reflejar cambios en la base de datos.
  - **ResultSet.TYPE_SCROLL_SENSITIVE**: Permite moverse en ambas direcciones y refleja los cambios en la base de datos.
- **Modos de Concurrencia**:
  - **CONCUR_READ_ONLY**: Solo permite lectura.
  - **CONCUR_UPDATABLE**: Permite la actualización de filas.

Ejemplo de ResultSet Bidireccional

```
String query = "SELECT * FROM usuarios";
try (Statement stmt =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
     ResultSet rs = stmt.executeQuery(query)) {
    rs.afterLast();
    while (rs.previous()) {
        System.out.println("Usuario: " + rs.getString("nombre"));
    }
}
```

---

6. Transacciones en JDBC

Las transacciones permiten agrupar múltiples operaciones SQL para que se ejecuten como una sola unidad. En caso de error, se puede hacer un rollback para deshacer todos los cambios.

Métodos Importantes

- **setAutoCommit(boolean autoCommit)**: Activa/desactiva el modo autocommit.
- **commit()**: Confirma la transacción.
- **rollback()**: Deshace la transacción.

Ejemplo de Transacción

```
try {
    connection.setAutoCommit(false);

    String updateSaldo = "UPDATE cuentas SET saldo = saldo - ? WHERE
id = ?";
    try (PreparedStatement pstmt =
connection.prepareStatement(updateSaldo)) {
        pstmt.setDouble(1, 500);
        pstmt.setInt(2, 1);
        pstmt.executeUpdate();
    }

    connection.commit();
} catch (SQLException e) {
    connection.rollback();
    e.printStackTrace();
}
```

---

7. Batch Processing

El procesamiento por lotes permite enviar múltiples sentencias SQL en una sola solicitud, optimizando el rendimiento en operaciones repetitivas.

Ejemplo de Batch con PreparedStatement

```
String insertQuery = "INSERT INTO usuarios (nombre, edad) VALUES (?,
?)";
try (PreparedStatement pstmt =
connection.prepareStatement(insertQuery)) {
    for (Usuario user : listaUsuarios) {
        pstmt.setString(1, user.getNombre());
        pstmt.setInt(2, user.getEdad());
        pstmt.addBatch();
    }
    pstmt.executeBatch();
}
```

---

8. Manejo de Excepciones con SQLException

La clase `SQLException` permite manejar errores de JDBC. Algunos métodos importantes incluyen:

- **getMessage()**: Muestra el mensaje de error.
- **getSQLState()**: Devuelve el estado SQL.
- **getErrorCode()**: Devuelve el código de error específico de la base de datos.

Ejemplo de Manejo de Excepciones

```
try {
    // Ejecución de consultas
} catch (SQLException e) {
    System.err.println("Error de SQL: " + e.getMessage());
    System.err.println("Estado SQL: " + e.getSQLState());
    System.err.println("Código de Error: " + e.getErrorCode());
}
```

---

9. Cierre de Recursos

Es importante cerrar los recursos JDBC (Connection, Statement y ResultSet) para evitar fugas de memoria. El bloque `try-with-resources` simplifica este proceso.

```
try (Connection conn = DriverManager.getConnection(url, user,
password);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(query)) {
    // Procesamiento
} catch (SQLException e) {
    e.printStackTrace();
}
```

Resumen Visual

- **Arquitectura JDBC**: Componentes esenciales para la conexión y manipulación de bases de datos.
- **Tipos de Statements**: Statement, PreparedStatement y CallableStatement.
- **Transacciones**: Controlan el conjunto de operaciones SQL como unidad.
- **Batch Processing**: Optimiza operaciones masivas.
- **Manejo de Excepciones**: Importante para identificar y solucionar errores de base de datos.
- **Cierre de Recursos**: Uso de `try-with-resources` para manejo seguro de recursos.

Este esquema avanzado sobre JDBC cubre conceptos fundamentales y técnicas necesarias para trabajar de forma eficiente con bases de datos en Java.