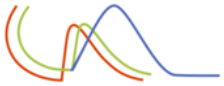


Contenido

JDBC (Java Database Connectivity) es una API de Java que proporciona acceso universal a datos desde el lenguaje de programación Java.

Con la API de JDBC, que proporciona una interfaz común, se puede acceder a prácticamente cualquier fuente de datos, desde bases de datos relacionales hasta hojas de cálculo y archivos sin formato. La API de JDBC se compone de dos paquetes:

- java.sql.
- javax.sql.

Ejemplo: uso de la misma interfaz para distintos tipos de bases de datos.

```
Connection conexion;
String driver = "com.mysql.jdbc.Driver";
String url = "jdbc:mysql://localhost:3306/";
String nombreBD = "universidad";
String user = "pruebas";
String pass = "pruebas";
// CONEXIÓN CON SGBD
conexion = DriverManager.getConnection(url + nombreBD, user, pass);
// PREPARACIÓN CONSULTA
Statement stmt = conexion.createStatement();
String select = "SELECT * FROM asignaturas;";
// EJECUCIÓN CONSULTA
ResultSet rs = stmt.executeQuery(select);
```

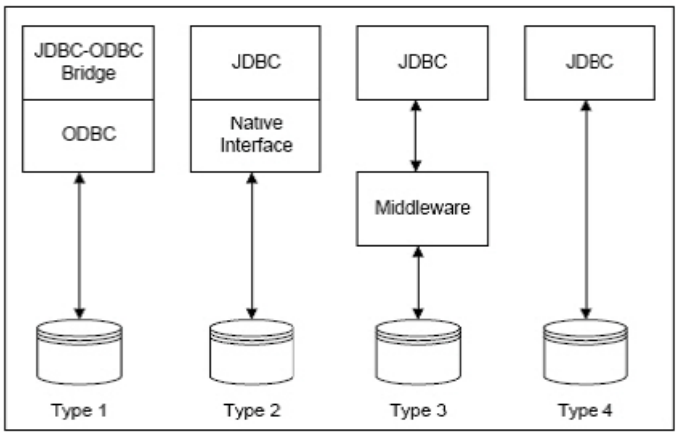
Para utilizar la API de JDBC con un SGBD concreto, se necesita un driver basado en tecnología JDBC como intermediario entre la tecnología JDBC y la base de datos.

Conviene resaltar que el paquete java.sql (API de JDBC) está formado principalmente por un conjunto de interfaces y que serán los drivers JDBC los encargados de implementar esos métodos.

Así pues, para establecer una conexión con un SGBD específico, como Oracle o MySQL, será el driver específico el que implemente las interfaces para la conexión con el SGBD y para interactuar con él. Por tanto, se requiere la instalación del driver concreto en la máquina que se ejecuta la app de Java.

Existen 4 tipos de drivers JDBC:

- Tipo 1 Puente JDBC-ODBC: el driver convierte las llamadas a métodos a la API de ODBC.
- Tipo 2 API nativa: el driver convierte las llamadas a métodos de la API específica de la base de datos.
- Tipo 3 Middleware: se utiliza un servidor intermedio entre el driver y la base de datos.
- Tipo 4 Java puro: el driver convierte las llamadas directamente al protocolo específico de la base de datos.



Driver Tipo 1 Bridge JDBC-ODBC

La API de Java traduce las llamadas a la tecnología ODBC. Se requiere tener instalado el driver ODBC en la máquina cliente.

- Ventajas: acceso a casi todos los SGBD ya que los drivers ODBC están muy extendidos.
- Desventajas: ineficiente al existir varios niveles de software, no recomendado si el número de transacciones es elevado.

Driver Tipo 2 API nativa

No se requiere de un driver ODBC. Las llamadas JDBC se traducen en llamadas específicas del API de la base de datos. Cada cliente debe tener instalado el driver.

- Ventajas: más eficiente que el driver Tipo 1.
- Desventajas: ligado al SGBD elegido.

Driver Tipo 3 Middleware

El driver JDBC convierte las llamadas a un protocolo independiente de la base de datos y las envía al servidor intermedio (middleware). Este servidor, que es una capa de abstracción, traslada las peticiones al SGBD concreto.

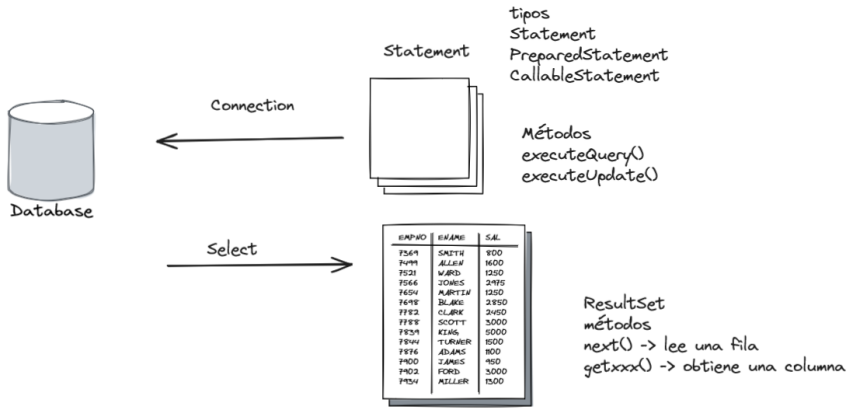
- Ventajas: el driver no depende del SGBD específico, con un solo driver se puede manejar cualquier base de datos (siempre que el middleware lo admita).
- Desventajas: el middleware añade un nivel más de software disminuyendo el rendimiento.

Driver tipo 4 Java puro

El driver nativo, desarrollado 100% en Java, es específico para cada SGBD. Así pues, el fabricante del SGBD proporcionará su driver JDBC de tipo 4. La llamada JDBC se traduce directamente en una a la base de datos, sin intermediarios.

- Ventajas: muy eficiente al no existir varios niveles de software.
- Desventajas: ligado al SGBD elegido.

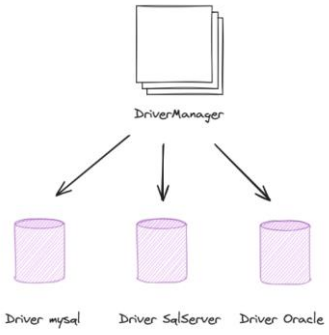
El paquete java.sql se compone de varias intefaces y clases que se pueden consultar en el siguiente enlace https://docs.oracle.com/en/java/javase/17/docs/api/java.sql/java/sql/package-summary.html .



ABRIR Y CERRAR CONEXIONES

Para poder acceder a una base de datos es necesario establecer una conexión. Antes de conectarnos con la base de datos hay que registrar el controlador apropiado.

Java utiliza la clase DriverManager para cargar inicialmente todos los controladores JDBC disponibles y que deseemos utilizar.



Para cargar un controlador se emplea del método forName() de la clase Class que devuelve un objeto Class asociado con la clase que se le pasa como parámetro.

La interfaz Connection es la que se encarga de la conexión con la base de datos. Mediante el método getConnection(), obtenemos un objeto de la clase Connection.Esta clase contiene todos los métodos que nos permiten manipular la base de datos.

Para cerrar la conexión utilizamos el método close() de la interfaz Connection.

```java
public class Conexion {  3 usages  1 inheritor  new *
    //atributos
    protected Connection miConexion;  4 usages
    //métodos
    public void abrirConexion() throws SQLException {  1 usage  new *
        //1. Registrar el driver
        try {
            Class.forName( className: "org.postgresql.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Error al registrar el driver de PostgreSQL: " + ex);
        }
        //2. Obtener la conexion
        miConexion = DriverManager.getConnection( url: "jdbc:postgresql://127.0.0.1:5432/estacion_autobuses",
                user: "postgres", password: "postgres");
        System.out.println("La conexion se ha abierto con exito");
    }

    public void cerrarConexion() throws SQLException {  no usages  new *
        miConexion.close();

    }
```

CREAR CONSULTAS CON SQL

Para crear una consulta a la base de datos podemos utilizar 3 interfaces diferentes, cada uno tiene un uso especifico:

- Statement; Para ejecutar querys completas, es decir, tenemos todos los datos
- de la query.
- PreparedStatement; Para ejecutar querys parametrizadas.
- CallableStatement; Para ejecutar querys a través de procedimientos almacenados definidos en la BBDD.

Statement

Para incluir una sentencia SQL utilizaremos la interfaz Statement, con el método createStatement() de la interfaz Connection.

Al crear una instancia del objeto Statement, podremos realizar sentencias SQL sobre la base de datos. Existen dos tipos de sentencias a realizar:

Sentencias de modificación (update); Engloban a todos los comandos SQL que

no devuelven ningún tipo de resultado como pueden ser los comandos INSERT, UPDATE, DELETE o CREATE.

Sentencias de consulta (query); Son sentencias del tipo SELECT (sentencias de consulta) que retornan algún tipo de resultado.

Para las sentencias "update" la clase Statement nos proporciona el método siguiente que devolverá el número de filas afectadas por la sentencia SQL:

```
public abstract int executeUpdate(String sentenciaSQL)
```

```java
public int insertar(Producto p) throws SQLException {  no usages  GabrielaGP2023
    //1. Declarar variables
    Statement comando;
    int resultado;
    String sql = "insert into products (product_id,product_name, discontinued) " +
            "values ( " + p.getIdProducto() +", '" +
            p.getNombreProducto()+ "', 1);";
    //2. Abrir conexion
    abrirConexion();
    //3. Obtener el Statement de la conexion
    comando = miConexion.createStatement();
    resultado = comando.executeUpdate(sql);
    comando.close();
    cerrarConexion();
    //4. devolver el resultado
    return resultado;
}
```

Para las sentencias "query" la clase Statement utiliza el método siguiente:

`public abstract ResultSet executeQuery(String sentenciaSQL)`

```java
public Producto consultarUno(int id) throws SQLException {  1 usage  GabrielaGP2023
    Statement comando;
    ResultSet rejilla;
    String sqlSentencia = "Select * from products where product_id = "+ id;
    Producto p1;
    //1. Abrir la conexion
    abrirConexion();
    //2. Creo el statement - se obtiene de la conexión
    comando = miConexion.createStatement();
    //3. Ejecuto la sentecia
    rejilla = comando.executeQuery(sqlSentencia);
    //4. Verificar si hay resultado
    if(rejilla.next()==true){
        //Se pudo leer una fila
        int idProducto;
        String nombreProducto;
        double precio;
        double cantidadExistencia;
        idProducto = rejilla.getInt( columnLabel: "product_id");
        nombreProducto = rejilla.getString( columnLabel: "product_name");
        precio = rejilla.getDouble( columnLabel: "unit_price");
        cantidadExistencia = rejilla.getDouble( columnLabel: "units_in_stock");
        p1 = new Producto(idProducto,nombreProducto,precio,cantidadExistencia);
    }else{
        //cuando no leyó una fila - significa que no hay datos en BBDD
        p1 = null;
    }
    //5. obtener cada valor de las columnas
    //6. devolver el resultado
```

PreparedStatement

También se usan para poder utilizar instrucciones SQL parametrizadas. Para establecer parámetros en una instrucción SQL basta con sustituir el dato por un signo de interrogación. (Select * from Tabla where Nombre=?).

El parámetro se sustituye por el valor mediante el método setXXX(int lugar, XXX Valor).

PreparedStatement no utiliza los métodos de Statement para ejecutar las querys, en su lugar, se pasa la query en el propio constructor.Igual que antes podemos ejecutar dos tipos de consultas query y update.

```
public int insertarP(Empleado e) throws SQLException {  1 usage   GabrielaGP2023
    //1. Declarar variables
    PreparedStatement comando;
    int resultado;
    String sql = "insert into employees " +
            "(employee_id, first_name, last_name) " +
            "values ( ?,?,?);";
    //2. Abrir conexion
    abrirConexion();
    //3. Obtener el Statement de la conexion
    comando = miConexion.prepareStatement(sql);
    //Dar valor a los parámetros
    comando.setInt( parameterIndex: 1,e.getId());
    comando.setString( parameterIndex: 2,e.getNombre());
    comando.setString( parameterIndex: 3,e.getApellido());
    resultado = comando.executeUpdate();
    comando.close();
    cerrarConexion();
    //4. devolver el resultado
    return resultado;
}
```

CallableStatement

Los objetos de este tipo se crean a través del método de Connection, prepareCall(). Se pasa como argumento de este método la llamada al procedimiento almacenado. Los CallableStatement también pueden ser parametrizados.

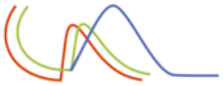
Un procedimiento almacenado es un objeto registrada en la base de datos y que realiza operaciones de cualquier tipo.

PROCESAR DATOS

Para procesar los datos recibidos tras ejecutar la query depende del tipo de consulta que hemos lanzado.

• Recordamos que si era una query de consulta devolvía un objeto de tipo ResultSet.

• Si era una query de modificación entonces recogemos una valor entero que será el número de registros modificados



La interfaz ResultSet contendrá las filas o registros obtenidas mediante la ejecución de la sentencia de tipo "query". Cada una de esas filas obtenidas se divide en columnas.

La interfaz ResultSet contiene un puntero que está apuntando a la fila actual. Inicialmente está apuntando por delante del primer registro. Para avanzar el puntero utilizamos el método next().

Una vez posicionados en una fila concreta, podemos obtener los datos de una columna específica utilizando los métodos getxxx() que proporciona la interfaz ResultSet, la "xxx" especifica el tipo de dato presente en la columna.

Para cada tipo de dato existen dos métodos getxxx():

getxxx(String nombreColumna); Donde especificamos el nombre de la columna donde se encuentra el dato.

• getxxx(int numeroColumna); Donde se especifica la posición de la columna

dentro de la consulta. Siempre **empezando desde 1**.

MANEJO DE TRANSACCIONES

Para poder manejar transacciones debemos utilizar el método setAutocommit para especificar si queremos un commit implícito o no.

Connection.setAutocommit(boolean);

• Si lo ponemos a true; estamos diciendo que se hace un commit implícito.

• Si se pone a false; no se realizará ninguna modificación en la base de datos hasta que explícitamente se haga un commit.

Si la conexión no hace commit implícito:

• connection.commit(): valida la transacción

• connection.rollback(): anula la transacción

METADATOS

Los metadatos se pueden considerar como información adicional a los datos que almacena la tabla.

Se utiliza un objeto ResultSetMetaData para recoger los metadatos de una consulta.

Dicho objeto expone los siguientes métodos:

getColumnName(); devuelve el nombre de la columna.

getColumnCount(); nos indica el numero de columnas de la consulta.

getTableName(); devuelve el nombre de la tabla.

getColumnType(); nos dice el tipo de datos de la columna.

isReadOnly(); especifica si los datos son de solo lectura.