CONEXIÓN CON BASES DE DATOS

API JDBC

JDBC (Java Database Connectivity) está compuesto por un número determinado de clases e interfaces que permiten a cualquier programa escrito en Java acceder a una base de datos.

Este conjunto de clases reside en los paquetes: java.sql y javax.sql

javax.sql añade funciones de servidor como los RowSet, los pools de conexiones o las transacciones.

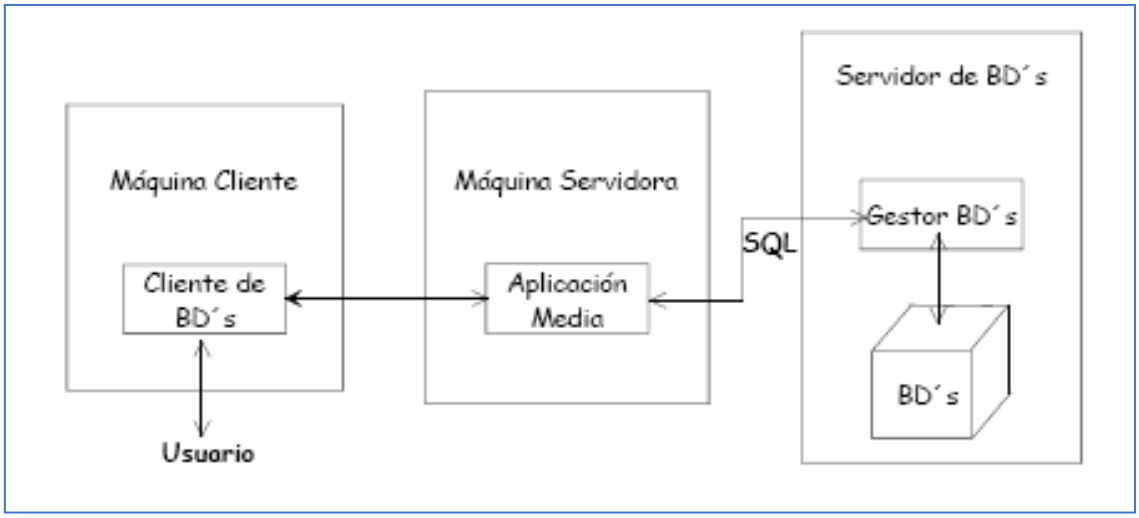

Gráfico 220. Esquema de conexión

Para podernos conectar a la base de datos necesitamos un controlador o también conocido como Driver. Este driver no es más que la implementación del API JDBC para un gestor de Base de datos concreto Oracle, MySQL, Derby, ...etc.

TIPOS DE CONTROLADORES

Los diferentes tipos de controladores que tenemos son:

- Puente JDBC-ODBC
- 100 % Java nativo
- 100 % Java / Protocolo independiente
- 100 % Java / Protocolo nativo

PUENTE JDBC-ODBC

Muchos servidores de BD's utilizan protocolos específicos para la comunicación. Esto implica que el cliente tiene que aprender un lenguaje nuevo para comunicarse con el servidor. Microsoft ha establecido una norma común para comunicarse con las bases de datos llamada Conectividad Abierta de Bases de Datos (ODBC). Hasta que esta norma no apareció, los clientes eran específicos del servidor. Utilizando la API ODBC lo que conseguimos es desarrollar software independiente del servidor.

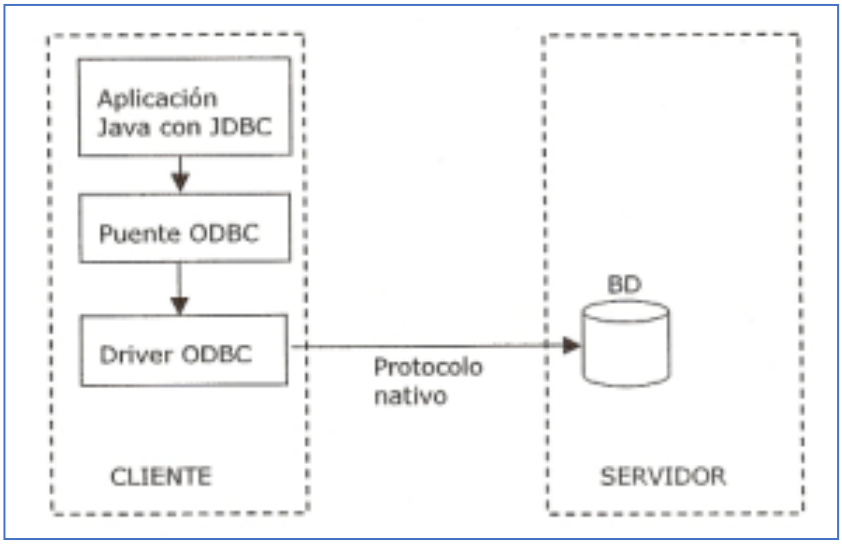

Gráfico 221. Conexión a través de ODBC

100 % JAVA NATIVO

Son controladores que usan el API de Java JNI (Java native interface) para presentar una interfaz Java a un controlador binario nativo del SGBD.

Su uso, igual que los anteriores, implica instalar el controlador nativo en la máquina cliente. Suelen tener un rendimiento mejor que los controladores escritos en Java completamente, aunque un error de funcionamiento de la parte nativa del controlador puede causar problemas en la máquina virtual de Java.

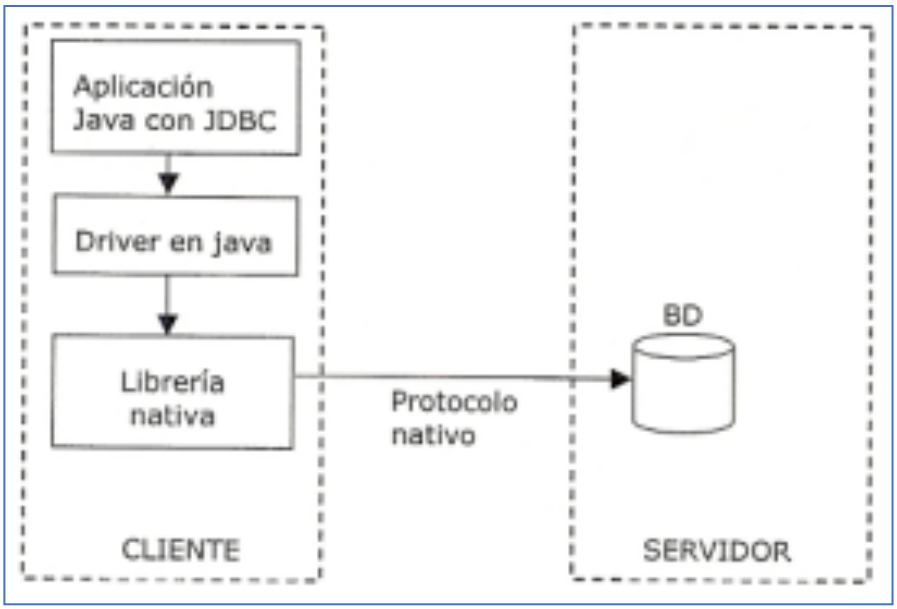

Gráfico 222. Conexión a través de Librería nativa

100 % JAVA / PROTOCOLO INDEPENDIENTE

Controladores escritos en Java que definen un protocolo de comunicaciones que interactúa con un programa de middleware que, a su vez, interacciona con un SGBD. El protocolo de comunicaciones con el middleware es un protocolo de red independiente del SGBD y el programa de middleware debe ser capaz de comunicar los clientes con diversas bases de datos.

El inconveniente de esta opción estriba en que debemos tener un nivel más de comunicación y un programa más (el middleware).

100 % JAVA / PROTOCOLO NATIVO

Son los controladores más usados en accesos de tipo intranet (los usados generalmente en aplicaciones web).

Son controladores escritos totalmente en Java, que traducen las llamadas JDBC al protocolo de comunicaciones propio del SGBD. No requieren ninguna instalación adicional ni ningún programa extra.

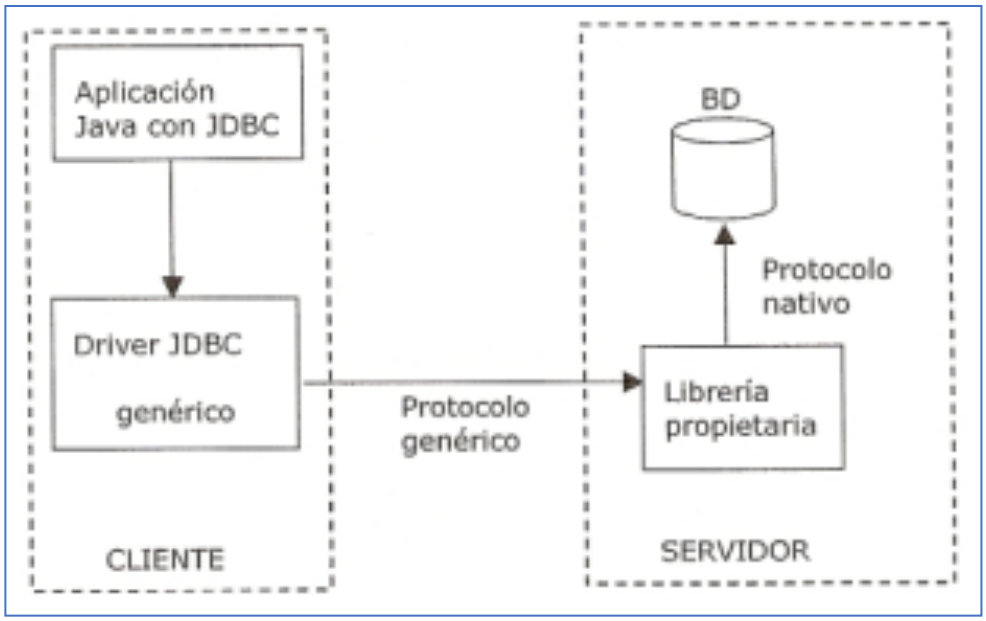

Gráfico 223. Conexión a través de protocolo nativo

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
private Connection conexion;

public void abrirConexion() {
    try {
        // Cargar el driver de Derby
        Class.forName("org.apache.derby.jdbc.ClientDriver");

        // Abrir la conexion a la BBDD
        conexion = DriverManager.getConnection("jdbc:derby://localhost:1527/Tienda",
                "curso", "curso");

    } catch (SQLException ex) {
        System.out.println("Error al abrir la conexion");
        ex.printStackTrace();
    } catch (Exception ex) {
        System.out.println("error al cargar el driver");
    }
}
```

Gráfico 224. Fragmento para abrir una conexión a Derby

```java
public void cerrarConexion() {
    try {
        // cerrar la conexion
        conexion.close();
    } catch (SQLException ex) {
        System.out.println("Error al cerrar la conexion");
    }
}
```

Gráfico 225. Fragmento para cerrar la conexión

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. No tenemos todos los datos en este momento y utilizamos parámetros.
- **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)

```
// 1.- Abrir la conexion
abrirConexion();

// 2.- Preparar la query
Statement stm = conexion.createStatement();

// 3.- Ejecutar la query
int resultados = stm.executeUpdate("delete * from APP.Productos");
```

Gráfico 226. Ejemplo executeUpdate

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

public abstract ResultSet executeQuery(String sentenciaSQL)

```
// 1.- Abrir la conexion
abrirConexion();

// 2.- Preparar la query
Statement stm = conexion.createStatement();

// 3.- Ejecutar la query
ResultSet resultados = stm.executeQuery("select * from APP.Productos");
```

Gráfico 227. Ejemplo executeQuery

PreparedStatement

Define métodos para trabajar con instrucciones SQL pre compiladas, que son más eficientes.

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.

```
// 1.- Abrir la conexion
abrirConexion();

// 2.- Preparar la query
String query = "insert into APP.Productos values (?,?,?) ";
PreparedStatement stm = conexion.prepareStatement(query);

// sustituimos los parámetros
stm.setInt(1, p.getId());
stm.setString(2, p.getDescripcion());
stm.setDouble(3, p.getPrecio());

// 3.- Ejecutar la query
int resultados = stm.executeUpdate();
```

Gráfico 228. Ejemplo PreparedStatement con executeUpdate

```
// 1.- Abrir la conexion
abrirConexion();

// 2.- Preparar la query
String query = "select * from APP.Productos where ID = ? ";
PreparedStatement stm = conexion.prepareStatement(query);

// asignar parámetros
stm.setInt(1, id);

// 3.- Ejecutar la query
ResultSet resultados = stm.executeQuery();
```

Gráfico 229. Ejemplo PreparedStatement con executeQuery

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 una macro registrada en la base de datos y que realiza operaciones de cualquier tipo.

```
// 1.- Abrir la conexion
abrirConexion();

// 2.- Preparar la query
String query = "{call procedimiento(?)}";
CallableStatement cstm = conexion.prepareCall(query);
cstm.setDouble(1, precio);

// 3.- Ejecutar la query
ResultSet resultados = cstm.executeQuery();
```

Gráfico 230. Ejemplo CallableStatement con executeQuery

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.

```
// 3.- Ejecutar la query
ResultSet resultados = cstm.executeQuery();

// 4.- Recorrer la coleccion de registros
while (resultados.next()) {
    lista.add(new Producto(resultados.getInt("ID"),
                           resultados.getString("DESCRIPCION"),
                           resultados.getDouble("PRECIO")));
}
```

Gráfico 231. Procesar resultados

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.

```
// 5.- Obtener los metadatos
ResultSetMetaData metadataRS = resultados.getMetaData();
int columnCount = metadataRS.getColumnCount();
for (int i = 0; i < columnCount; i++) {
    System.out.println("nombre columna " +
                            metadataRS.getColumnName(i));
}
```

Gráfico 232. Obtención de los metadatos de la consulta

Todo el código de este ejemplo lo encontrareis en **Ejemplo37_ JDBC.zip**

RECUERDA QUE...

- Con el API JDBC podemos acceder una base del datos sin importarnos el fabricante.
- JDBC permite abrir y cerrar conexiones, así como ejecutar consultas a la BBDD.