

Conectores a base de datos

Conector

Serie de **clases y librerías** que **unen** la capa de nuestra **aplicación** con la capa de **base de datos**. Este **punto intermedio** es nuestro conector, y es necesario para conectarnos a la base de datos y realizar consultas.

Desfase objeto-relacional

Se refiere a cuando **surgen discrepancias: bases de datos tienen naturalezas** distintas en comparación a la aplicación que se trabaja con programación **orientada a objetos**. Esto se llama: desfase objeto-relacional.

Aspectos importantes del desfase:

- **Diferencia entre los datos:** diferencias en los datos:

En la BDD **relacional** son **datos simples**;

En la **orientada a objetos** son **objetos complejos**.

- **Realizar una traducción:** Realizar distintos **diagramas**: traducción desde los objetos de la aplicación Java a la base de datos relacional. **Entidades distintas** representen la **misma unidad**.

Protocolos de acceso a base de datos

Realmente, un conector o **driver** es una serie de clases implementadas (API) que facilitan la **conexión a la base de datos** asociada.

Basándonos en el lenguaje **SQL**, disponemos de **dos protocolos** de conexión:

- **JDBC** (Java Database Connectivity) (**Sun**).
- **ODBC** (**Open** DataBase Connectivity) (**Microsoft**) *Basado en la conexión con bases de datos puras SQL. API desarrollada en **lenguaje C**.*

Otros protocolos:

De Microsoft también tenemos:

- **ADO.NET**
- **ADO.NET + LINQ**
- **OLE/ADO DB**

Una aplicación debe tener asociado siempre un conector. Cuando estamos desarrollando una aplicación e introducimos un conector no tenemos que conocer los aspectos técnicos, ni cómo funcionan en su interior dichas bases de datos, sino sólo en **cómo realizar la comunicación** y de **cómo funcione** nuestra aplicación.

El **conector interpretaría** de una forma u otra **dependiendo de la base de datos** asociada.

Si nuestra aplicación necesita información de una base de datos, **utilizando la librería correspondiente** e indicando las **configuraciones de acceso** a cada base de datos, tendremos el acceso sin preocuparnos del lenguaje interno de cada una.

Conexiones JDBC: Componentes y tipos

Componentes JDBC

Son **cuatro**:

- **API JDBC:**

- librerías y clases que nos facilitan:
 - 1- **acceso** a las bases de datos relacionales.
 - 2- **consultas** a la base de datos.

java.sql y **javax.sql**.

- **Paquete de pruebas JDBC:**

Valida si un **driver** pasa los **requisitos** previstos por JDBC.

- **Gestor JDBC:**

Realiza la **unión** (conexión) **aplicación** - **driver** apropiado JDBC.

Hay **dos formas** de conexión:

- **Directa.**
- Con **pool** de conexiones.

- **Puente JDBC-ODBC:** facilita el **uso de los drivers ODBC** como si estuviéramos trabajando con JDBC.

2 tipos de Arquitectura de conexión con JDBC

- **En dos capas:** nuestra aplicación se conectará a la BDD a través de **un driver**. Driver y aplicación estarán **en el mismo sistema o máquina**.

Será ideal para una **aplicación simple** que **no requiere muchos recursos** ni se prevé que vaya a tener multitud de consultas. Se puede instalar el **conector** (driver) en la **misma máquina del cliente** realizando las labores de **traducción** y comunicándose **directamente con la base de datos**.

Las capas serán:

- 1- la **aplicación junto con el driver** en el sistema o máquina
- 2- y la **base** de datos.

- **En tres capas:** La aplicación envía instrucciones a una **capa intermedia (driver)**, a modo de **traductor (middleware** o software intermedio). La capa intermedia o *middleware* **cogerá** la información y la **enviará** a la base de datos correspondiente **traduciendo los comandos** que la aplicación haya enviado. Es más aconsejada para **aplicaciones web** (e-commerce), en la que se aísla el driver del sistema que contiene la aplicación, **no** teniendo que hacer esta **traducción** de comandos **en el sistema** o máquina donde se hace la petición. De esta forma es **más rápido** el acceso a la base de datos y la respuesta a la aplicación en el lado del cliente (aplicación o sistema).

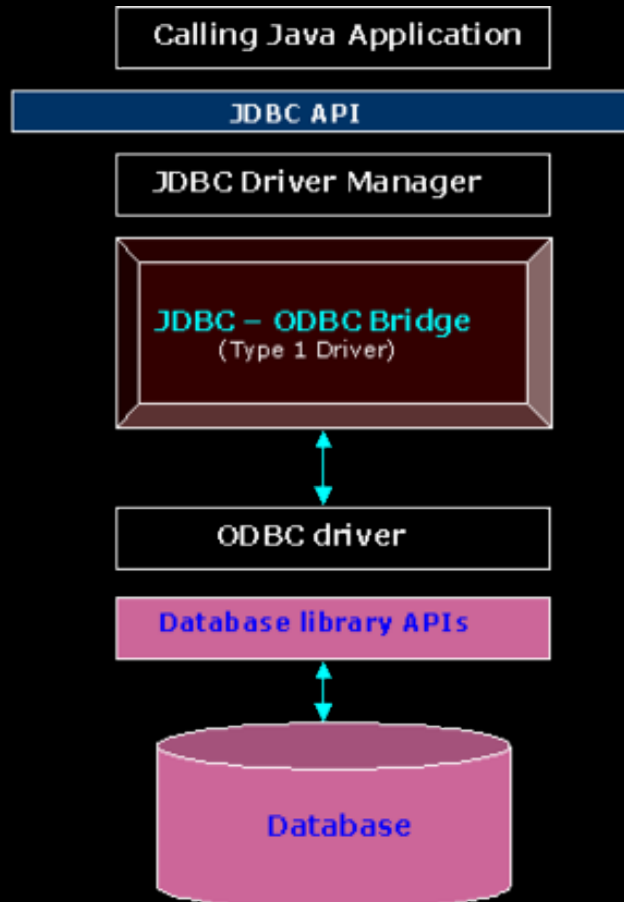
Útil para aplicación (escritorio o web) cuyo propósito sea gestionar una **cantidad grande de consultas** y sea necesario balancearlas, incluso con algún tipo de caché.

Las capas serán:

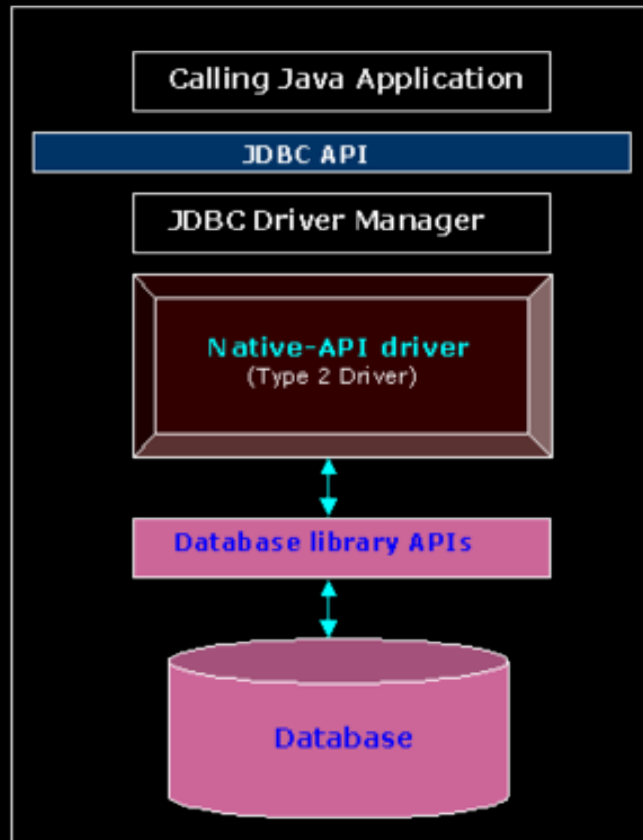
- 1- la **aplicación** en el sistema o máquina,
- 2- el **driver** *middleware* o *traductor*,
- 3- y la **base** de datos.

Tipos de conexiones JDBC

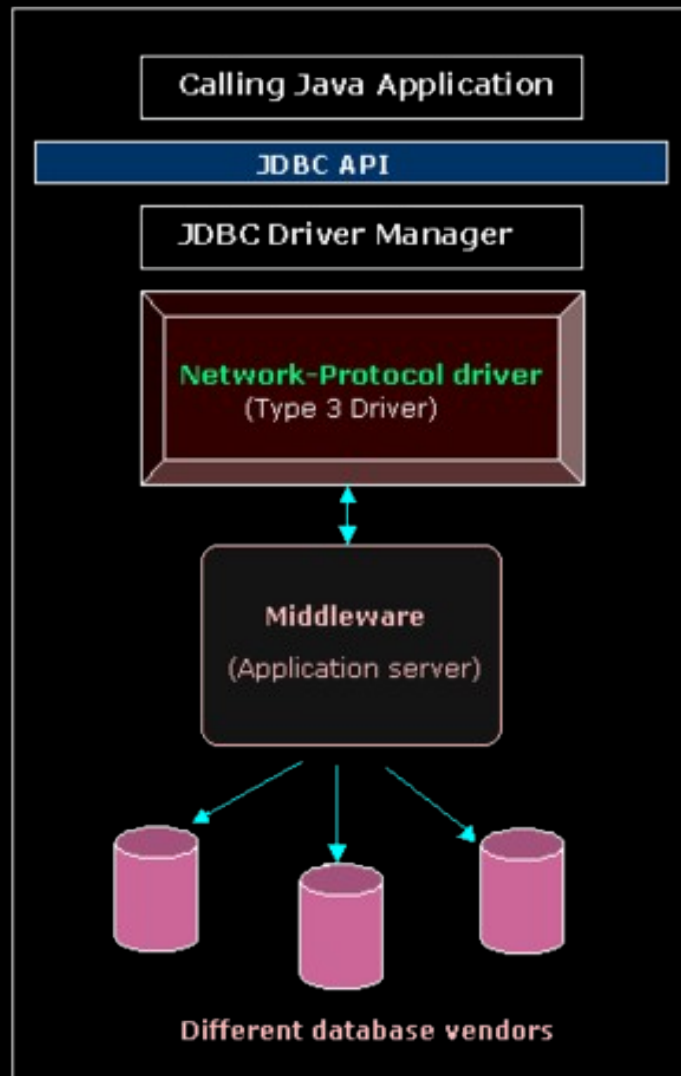
- **Driver tipo 1 JDBC-ODBC (Puente JDBC-ODBC)**: este driver usa una **API nativa**, **traduce** las llamadas realizadas de JDBC a ODBC. Los **datos devueltos** por la base de datos se **traducirán a JDBC** cuando sean devueltos.



- **Driver tipo 2 JDBC Nativo (driver API-Nativo):** estos drivers están escritos una parte en **Java** y otra parte, en código **nativo**. Las **llamadas al API JDBC** son traducidas en **llamadas propias** nativas de la **API de la base de datos** relacional que tengamos. **Más rápido** que el puente JDBC-ODBC pero se necesita instalar la librería cliente de la base de datos en la máquina cliente y el driver es dependiente de la plataforma.



- **Driver tipo 3 JDBC net:** **Middleware** entre el JDBC y el SGBD. Es de **tres capas** cuyas solicitudes JDBC están siendo **traducidas** en un protocolo de red en una capa intermedia (**middleware**). Esta capa intermedia recibirá dichas solicitudes y las enviará a la base de datos usando un driver **JDBC de tipo 1 o de tipo 2**. Es una arquitectura muy **flexible**.

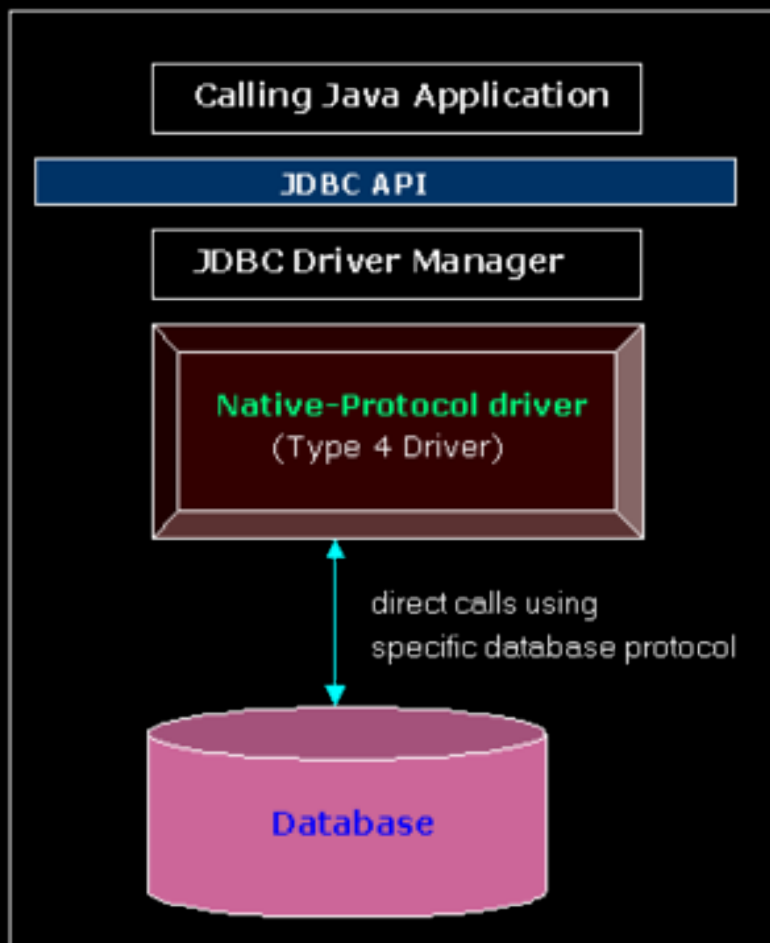


- **Driver tipo 4 protocolo nativo (controlador Java puro):** este tipo de driver realiza las llamadas **mediante el servidor**, usando el protocolo nativo del mismo. Estos drivers pueden desarrollarse al completo **en Java**. Si en el futuro se necesitara hacer un **cambio de base de datos**, evidentemente, habría que desarrollar **otro driver nativo** adaptado a la **nueva base** de datos relacional.

Wiki: El controlador JDBC tipo 4, también conocido como controlador Java puro directo a base de datos , es una implementación de controlador de base de datos que **convierte llamadas JDBC** directamente en un **protocolo** de base de datos **específico del proveedor**.

Escritos completamente en Java, los controladores tipo 4 son, por tanto, **independientes de la plataforma**. Se instalan **dentro de la máquina virtual Java** del cliente. Esto proporciona un mejor **rendimiento** que los controladores tipo 1 y tipo 2, ya que no tiene la sobrecarga de conversión de llamadas en ODBC o llamadas API de bases de datos. A diferencia de los controladores tipo 3, **no necesita software asociado** para funcionar.

Como el protocolo de la base de datos es específico del proveedor, el cliente JDBC requiere **controladores separados**, generalmente proporcionados por el proveedor, para conectarse a diferentes tipos de bases de datos.



Configuración de una conexión en código

Ejemplo de conexión en línea de código:

```
/**
 *1- El primer paso: sería descargar el driver (suele ser “.jar”) de conexión de la base
 de datos que vamos a utilizar ...
 2- y, a continuación, añadirlo a nuestro proyecto Java (aplicación).
 El primer paso para la conexión de una base de datos externa por medio de un driver de
 conexión es definir algunos literales que nos van a hacer falta, como el literal “Driver”,
 que hace referencia a la librería que hemos añadido a nuestro aplicativo, y la
 “URL CONEXION”, que hace referencia a la URL donde se alojará la información.
 Estas, podemos definir las como variables estáticas generales, ya que accederemos luego.
 */

private static final String DRIVER = “org.mysql.jdbc.Driver”;
private static final String URL_CONEXION = “jdbc:mysql://localhost:3306/Pruebas”;
```

Como estamos realizando una **prueba** de desarrollo, hemos introducido el código en nuestro **método main**. Normalmente se implementaría usando arquitectura DAO (**Data Access Object** o patrón de diseño en el cual una clase se encarga de las operaciones de persistencia en una tabla de la base de datos.).

```
public static void main(String args[]) throws SQLException {

    /**
     Se definen variables de tipo String que nos van a servir para realizar la conexión con la
     base de datos más tarde.
     Instanciamos el usuario y la contraseña de nuestra conexión y también una variable de
     tipo Connection y otra Statement.
     Normalmente se definen los literales de usuario y password en la capa DAO.
     */
    final String usuario = “user_db”;
    final String password = “password_db”;
    Connection dbConnection = null;
    Statement statement = null;
```

Connection es una **interfaz** que representa una **conexión directa** con una **base de datos**. El motivo de que sea una interfaz es porque tendrá **distintas implementaciones** posibles.

JDBC ofrece **distintas formas** para realizar **conexiones**. Nos centraremos en establecer la conexión con “java.sql.DriverManager”, recomendada para aquellas **aplicaciones** que se hayan **desarrollado en** lenguaje **Java**.

Establecer conexión

Podremos tener instaladas tantas conexiones como queramos. Cada **conexión** y cada **base de datos** utilizará los drivers JDBC, y, a su vez, cada uno de ellos implementará la **interfaz "java.sql.Driver"** . Con el método principal **connect()**, obtendremos el objeto Connection y **estableceremos la conexión** con base de datos.

Una vez que DriverManager nos ha devuelto la conexión a base de datos, realizaremos una **consulta simple** y la almacenaremos en una variable de tipo **String** para más tarde ser ejecutada.

```
try {  
    /**  
    * Registramos el driver que anteriormente hemos indicado en la variable estática "DRIVER".  
    * Con esta instrucción cargamos la librería "org.mysql.jdbc.Driver".  
    * Con Class.forName estaremos dando de alta un driver en nuestra aplicación:  
    * /  
    Class.forName(DRIVER);  
    /**  
    * El objetivo de la clase DriverManager, realmente, es gestionar los drivers que  
    * poseemos en nuestra aplicación y permitir en una misma capa el acceso a todos  
    * y cada uno de ellos. Algo que debemos tener en cuenta es que DriverManager  
    * necesita que todos y cada uno de los drivers estén registrados antes de su uso.  
    * Las conexiones deben quedar almacenadas antes de acceder a la base de datos.  
    * Después de haber registrado el driver, se pueden usar los métodos estáticos  
    * para hacer "getConnection", usándolo directamente para establecer conexiones.  
    * Al método "getConnection" le pasamos por parámetro la URL de conexión previamente  
    * definida: usuario y contraseña.  
    * Nos devolverá un objeto de tipo Connection, en nuestro caso lo hemos llamado  
    * dbConnection. De modo que en dbConnection tendríamos la conexión.  
    */  
    dbConnection = DriverManager.getConnection(URL_CONEXION, usuario, password);  
    /**  
    * Y ahora ya podemos usar la Base de Datos con sentencias, etc...  
    */  
    String selectTableSQL = "SELECT ID,USERNAME,PASSWORD,NOMBRE FROM Usuarios";  
    /**  
    * Y creamos el Statement (declaración SQL) en nuestra conexión a la BDD.  
    * El resultado de la petición a la BDD se almacenará en un ResultSet:  
    * Con la variable Connection, ejecutamos el método "createStatement" y lo asignamos  
    * a la variable definida al principio del ejercicio de tipo Statement.  
    * Realizamos la consulta con el método "executeQuery"  
    * pasándole como parámetro la query previamente definida en la variable de tipo String.  
    */  
    statement = dbConnection.createStatement();
```

```
/**
```

El resultado de la query se asignará a una variable de tipo **ResultSet** (rs).

La **lectura** del ResultSet está envuelto en un bucle “**while**”,

ya que por cada **fila** que nos devuelva esta **tabla**, podremos ir dando una **vuelta** más al **bucle** y seguir mostrando los resultados.

Mostraremos por pantalla tanto el **ID**, el **USERNAME**, el **PASSWORD** y el **NOMBRE**, que son **columnas** de la **tabla Usuarios** que hemos consultado de prueba.

```
*/
```

```
ResultSet rs = statement.executeQuery(selectTableSQL);
```

```
while (rs.next()) {
```

```
    String id = rs.getString("ID");
```

```
    String usr = rs.getString("USERNAME");
```

```
    String psw = rs.getString("PASSWORD");
```

```
    String nombre = rs.getString("NOMBRE");
```

```
    System.out.println("userid : " + id);
```

```
    System.out.println("usr : " + usr);
```

```
    System.out.println("psw : " + psw);
```

```
    System.out.println("nombre : " + nombre);
```

```
}
```

Operaciones con variables y excepciones

```
} catch (SQLException e) {  
    /**  
     * Excepción capturada si a la hora de ejecutar el método "executeQuery"  
     * algo va mal en base de datos, ya sea gramaticalmente, sintácticamente, etc.  
     */  
    System.out.println(e.getMessage());  
}  
catch (ClassNotFoundException e) {  
    /**  
     * Excepción lanzada y capturada en este punto si en nuestra línea:  
     * "Class.forName(DRIVER)"  
     * el fichero del driver que le estamos indicando no encontrara la librería.  
     */  
    System.out.println(e.getMessage());  
}  
finally {  
    /**  
     * La sentencia finally se ejecutará siempre, hayamos capturado excepción o no.  
     * En esta, simplemente, se realizan los cierres de la clase Statement y del  
     * objeto Connection que, a su vez, en este punto pueden lanzar una excepción  
     * que será recogida y lanzada a la capa superior a través de la palabra clave  
     * "Throws" en la definición de nuestro método.  
     */  
    if (statement != null) {  
        statement.close();  
    }  
    if (dbConnection != null) {  
        dbConnection.close();  
    }  
}
```

Ventajas e inconvenientes del uso de conectores

Drivers tipo 1 (Puente JDBC-ODBC)

- **Ventajas:**

- Solemos encontrarlos fácilmente, ya que se distribuyen **con el paquete** del lenguaje Java.
- Acceso a gran cantidad de **drivers ODBC**.

- **Inconvenientes:**

- Rendimiento: **demasiadas capas** intermedias.
- Limitación de **funcionalidad**. (Características comunes de base de datos).
- No funcionan bien con **applets**. **Problemas en navegadores**.

Drivers tipo 2 (driver API-Nativo)

- **Ventajas:**

- Ofrecen **rendimiento superior** al de tipo 1, ya que son llamadas **nativas**.

- **Inconvenientes:**

- La **librería** de la BDD, forzosamente, se inicia en la parte de **cliente**. No se pueden usar en internet.
- **Interfaz** nativa **Java**. No movable entre plataformas.

Drivers tipo 3 (JDBC net - Middleware)

- **Ventajas:**

- **No necesita librería del fabricante**. No es necesario llevar al cliente este aspecto.
- Son los que mejor **rendimiento** dan en **internet**, muchas opciones de **portabilidad** y **escalabilidad**.

- **Inconvenientes:**

- Requieren de un **código específico** de BDD para la **capa intermedia**.

Drivers tipo 4 (controlador Java puro)

- **Ventajas:**

- Buen **rendimiento**.
- No **necesitan instalar un software especial** ni en la parte del servidor, ni en la parte de cliente.
Drivers de fácil acceso.

- **Inconvenientes:**

- El usuario necesitará **distinto** software de conexión (**driver**) para cada **base de datos**.

Ejemplo de clase que realice las gestiones de conexión, e instancie cualquiera de los **parámetros** que se les vayan pasando de los diferentes **drivers**.

Implementar en esa clase un método **getDBConnection()** en donde deberá pasar cuatro atributos por parámetro:

- Nombre del driver
- Url de conexión
- Usuario
- Contraseña

El método **getDBConnection()** podría ser así:

```
private Connection getDBConnection(String Driver, String url, String usuario, String password)
{
    Connection connection = null;
    try {
        Class.forName(Driver);
        connection= DriverManager.getConnection(url, usuario, password);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return connection;
}
```

Se podrán **instanciar tantas conexiones** con bases de datos **como queramos**.

Recuerda:

Carga del driver

```
private static final String DRIVER = "nombre_del_Driver";  
private static final String URL_CONEXION = "url_de_la_base_De_datos";
```

Almacenamiento de credenciales:

```
final String usuario = "usuario";  
final String password = "contraseña";
```

Abrir la conexión:

```
Class.forName(DRIVER);  
  
dbConnection = DriverManager.getConnection(URL_CONEXION, usuario, password);
```

En un **método**, **podemos devolver la conexión** ya establecida con nuestra URL de conexión, usuario y contraseña configurado y preparado para trabajar con la base de datos:

```
Class.forName(DRIVER);  
  
return DriverManager.getConnection(URL_CONEXION, usuario, password);
```