Bases de datos con JDBC

Sesión 1: Introducción a JDBC

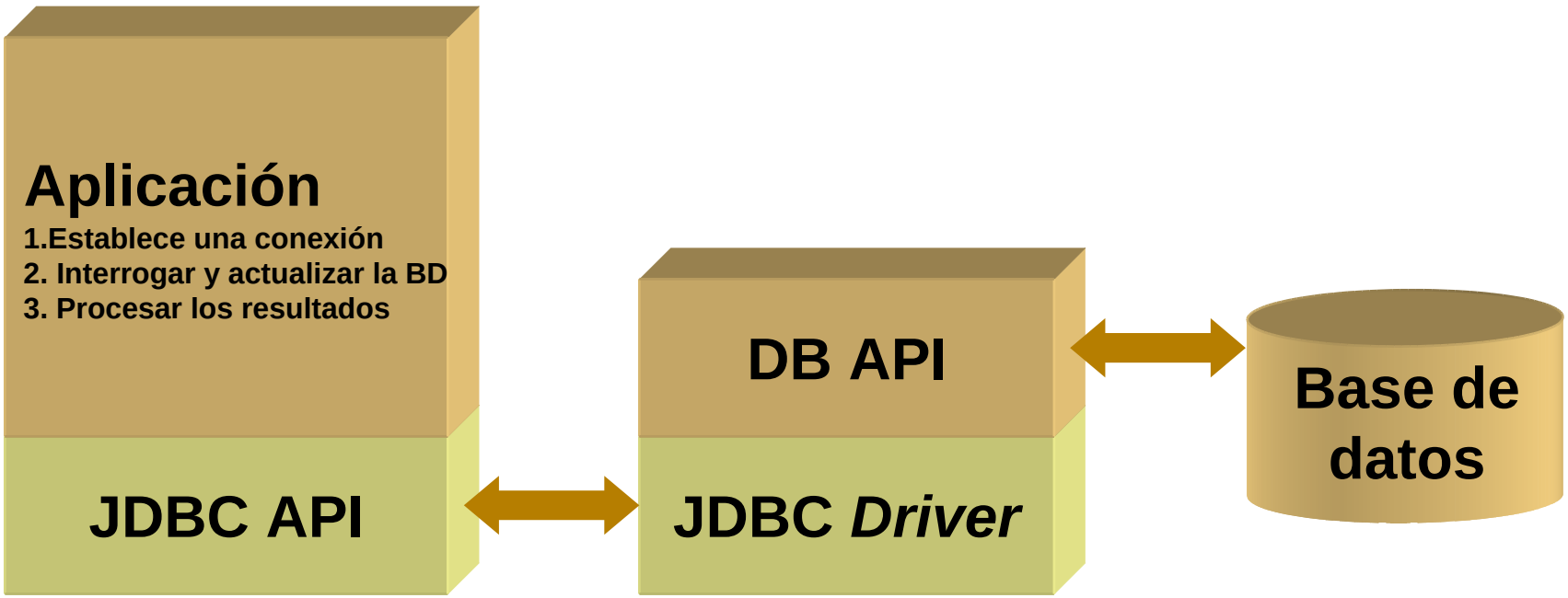
Índice

- Introducción
- *Drivers* de acceso a bases de datos
- Conexión con la base de datos
- Consulta a una base de datos

Introducción

- JDBC es el API de Java para acceder a sistemas de gestión de bases de datos (SGBD)
- Al hacer uso del API nos va a permitir cambiar de SGBD sin modificar nuestro código
- JDBC es una especie de “puente” entre nuestro programa Java y el SGBD

Esquema de uso de JDBC

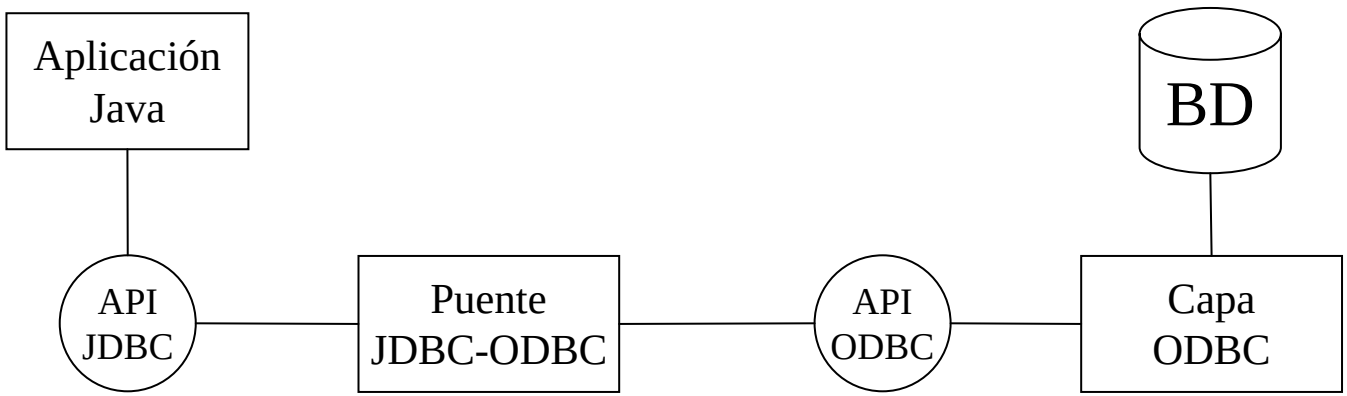


Drivers de acceso

- Para acceder a una BD necesitamos un *driver* específico
- Cada BD suele disponer de un API de acceso propietario
- Si usamos ese API, un cambio en la BD provocaría cambios en nuestro código
- El *driver* es específico para esa BD, al cambiar la BD sólo tenemos que cambiar el *driver*
- El *driver* traduce la llamada JDBC en la correspondiente llamada al API de la BD

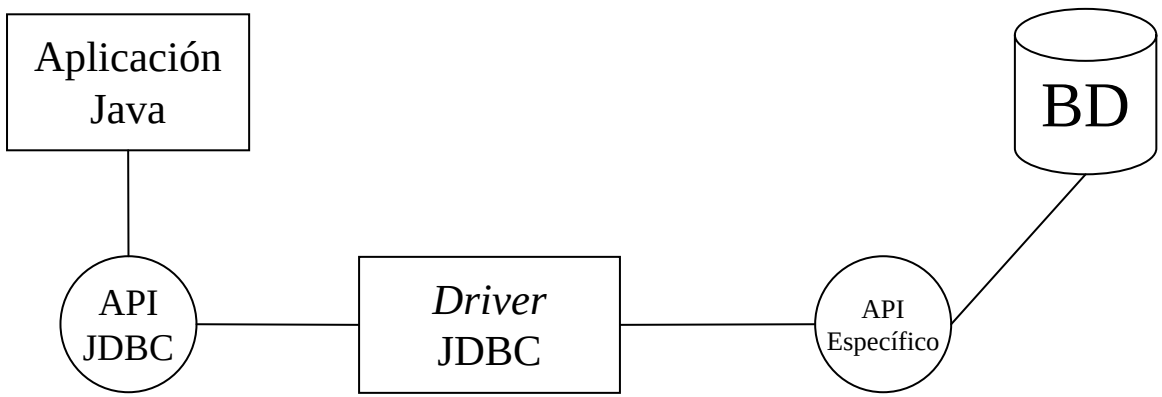
Tipos de *drivers*

- Tipo 1: Puente JDBC-ODBC
  - Proporciona conectividad entre Java y cualquier base de datos en Microsoft Windows, mediante ODBC
  - No se aconseja su uso. Limita las funcionalidades de las BD
  - Cada cliente debe tener instalado el *driver*
  - J2SE incluye por defecto este *driver* (Windows y Solaris)



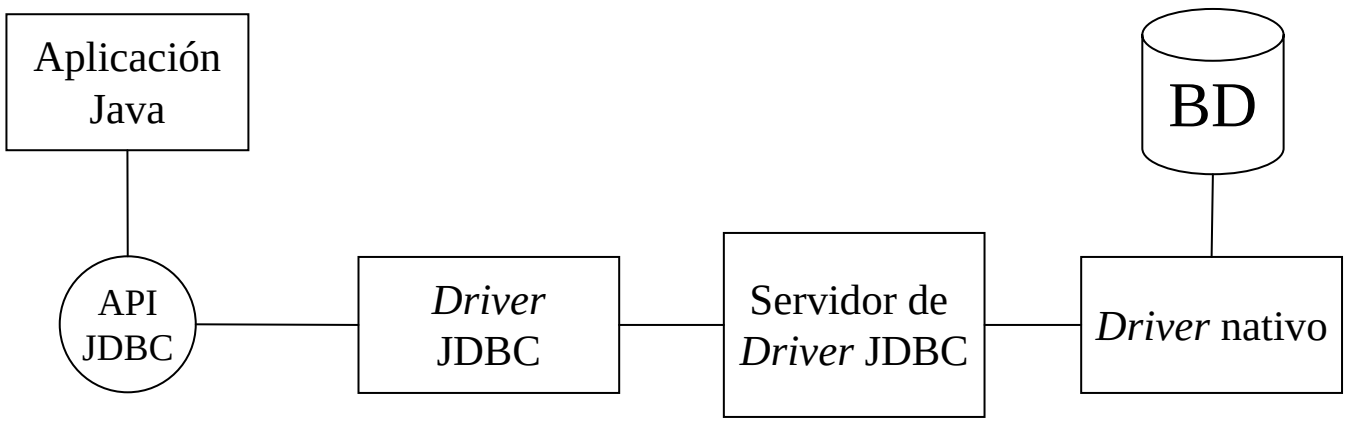
Tipos de *drivers*

- Tipo 2: Parte Java, parte *driver* nativo
  - El *driver* actúa como traductor de la llamada Java a una llamada del API de la BD. Necesita el API de forma local (no usar en Internet)
  - Es un paso menos que el anterior, pues no tenemos que pasar por el gestor ODBC (más rápido)
  - Cada cliente necesita el *driver*



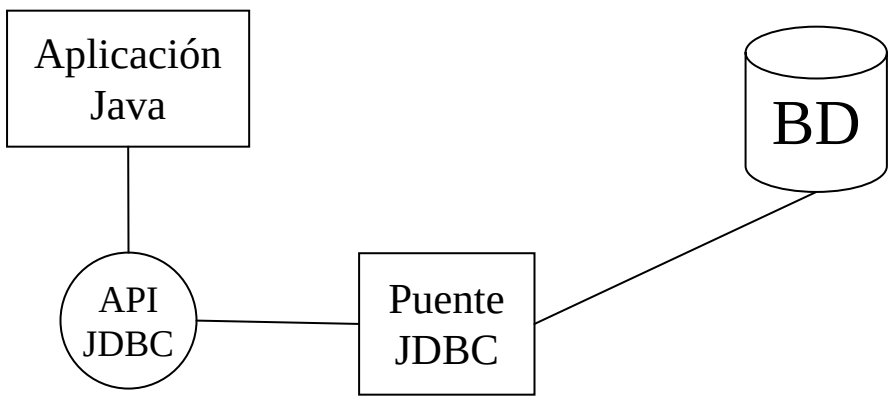
Tipos de *drivers*

- Tipo 3: Servidor intermediario de acceso a BD
  - Proporciona una mayor abstracción
  - Dispondremos de un componente servidor intermedio, que gestiona la conexión con una o varias BD
  - WebLogic implementa este *driver*
  - Útil para aplicaciones escalables y portables



Tipos de *drivers*

- Tipo 4: *Drivers* Java
  - El más directo
  - La llamada JDBC se traduce en una llamada a la propia BD, por la red y sin intermediarios
  - Mejor rendimiento
  - La mayoría de SGBD disponen de este *driver*



Sobre los distintos tipos

- Podemos disponer de *drivers* de distinto tipo para acceder a la misma BD
- Por ejemplo, MySQL desde su propio *driver* y desde ODBC
- Debemos tener en cuenta que un tipo de *driver* puede limitar las funcionalidades de la BD. En este caso, si utilizamos ODBC no tendremos acceso al control de transacciones de MySQL
- Resumiendo, utilizar siempre el *driver* del fabricante

Instalación de *drivers*

- Descargamos el *driver* específico para nuestra BD (normalmente es un .jar)
- Lo añadimos al CLASSPATH
  - export CLASSPATH=$CLASSPATH:/dir-donde-este/fichero
- Lo cargamos de forma dinámica dentro de nuestro código Java:
  - MySQL: Class.forName("com.mysql.jdbc.Driver"); Podéis encontrar también la clase org.gjt.mm.mysql.Driver
  - PostGres: Class.forName("org.postgresql.Driver");
  - ODBC: Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
- Se deben capturar las excepciones SQLException

Conexión a la BD

- Primero debemos conectarnos con la BD
  - Connection con = DriverManager.getConnection(url);
  - Connection con = DriverManager.getConnection(url, login, password);
- El url cambiará de una BD a otra, pero todas mantendrán el siguiente formato:
  - jdbc:<subprotocolo>:<nombre>
  - *jdbc* siempre
  - *subprotocolo* es el protocolo a utilizar.
  - *nombre* es la dirección (o el nombre) de la BD

Ejemplos de conexiones

- MySQL
  - Connection con = DriverManager.getConnection("jdbc:mysql://localhost/bd", "miguel", "m++24");
- PostGres
  - Connection con = DriverManager.getConnection("jdbc:postgresql://localhost:5432/bd", "miguel", "m++24");
- ODBC
  - Connection con = DriverManager.getConnection("jdbc:odbc:bd");

DriverManager

- Este objeto gestiona todo el paso de información con el driver
- Algunos métodos útiles de esta clase:
  - DriverManager.setLogWriter(new PrintWriter(System.out, true)); *// Muestra por la salida estándar cualquier operación que se realice con el driver*
  - DriverManager.println("Esto es un mensaje"); *// Nos permite depurar nuestro código*

Consulta a una BD

- La conexión a la BD la podemos utilizar para consultar, insertar o borrar datos
- Todas estas operaciones se realizarán mediante SQL
- La clase *Statement* nos permitirá realizar estas acciones
- Para crear un objeto de esta clase
  - Statement stmt = con.createStatement();

Consulta (*Query*)

- Para consultar datos utilizamos el método *executeQuery* de la clase *Statement*
  - ResultSet result = stmt.executeQuery(query);
- *Query* es un *String* que contiene la sentencia SQL
- La llamada al método nos devuelve un objeto de la clase *ResultSet*
- La respuesta es una tabla que contendrá una serie de campos y unos registros, dependiendo de la consulta realizada

Ejemplo de consulta

- String query = "SELECT * FROM ALUMNOS WHERE sexo = 'M'";
- ResultSet result = stmt.executeQuery(query);

- Imaginemos que la tabla ALUMNOS tiene tres campos, el resultado almacenado en *result* es

Registro

exp	nombre	sexo
1286	**Amparo**	**M**
1287	**Manuela**	**M**
1288	**Lucrecia**	**M**

Acceso a los valores de *ResultSet*

- La clase *ResultSet* dispone de un cursor que nos permite movernos por los registros
- Cuando ejecutamos la llamada, el cursor está en la posición anterior al primer registro
- Para mover el cursor a la siguiente posición utilizaremos el método *next* de *ResultSet*
- *Next* devuelve cierto si ha conseguido pasar al siguiente registro y falso si se encuentra en el último
- Para acceder a los datos del *ResultSet*, haremos un bucle como este:
  - while(result.next()) {
    // Leer registro
    }

Obtención del valor de los campos

- El cursor está situado en un campo
- Para obtener los valores de los campos utilizaremos los métodos *getXXXX(campo)* donde *XXXX* es el tipo de datos Java de retorno
- El tipo de datos del campo debe ser convertible al tipo de datos Java especificado
- El campo se especifica mediante un String o mediante un índice entero, cuyo valor dependerá de la consulta realizada
- *No se puede acceder dos veces al mismo campo*

Tipos de datos

- Los principales métodos que podemos utilizar son:

getInt	**Datos enteros**
getDouble	**Datos reales**
getBoolean	**Campos booleanos (si/no)**
getString	**Campos de texto**
getDate	**Tipo fecha (devuelve *Date*)**
getTime	**Tipo fecha (devuelve *Time*)**

Ejemplo

```
int exp;
String nombre;
String sexo;

while(result.next()) {
    exp = result.getInt("exp");
    nombre = result.getString("nombre");
    sexo = result.getString("sexo");
    System.out.println(exp + "\t" + nombre + "\t" +
sexo);
}
```

Posible problema

- Si el campo a consultar no contiene ningún valor, la llamada a *get* devuelve 0, si es número, y *null* si es un objeto
- En el caso de *nombre* si intentamos realizar una llamada a algún método de la clase String, se producirá una excepción
- Para evitar esto, podemos llamar al método *wasNull*(), que devuelve cierto si el último registro consultado no tenía un valor asignado

Ejemplo

- String sexo;

```
while(result.next()) {
    exp = result.getInt("exp");
    System.out.print(exp + "\t");
    nombre = result.getString("nombre");
    if (result.wasNull())
        System.out.print("Sin nombre asignado");
    else
        System.out.print(nombre.trim());
    sexo = result.getString("sexo");
    System.out.println("\t" + sexo);
}
```

¿Preguntas...?