

Tema 8

Acceso y gestión de bases de datos relacionales mediante JDBC

JDBC. Qué es

Es una API para la ejecución de operaciones sobre bases de datos relacionales desde Java, independientemente del sistema operativo donde se ejecute o del SGBD accedido. Permite, pues, que Java pueda acceder a cualquier base de datos relacional (Oracle, MySQL, Sybase, Informix, Access, SQLServer etc.) de forma transparente al tipo de la misma. JDBC está formado por un conjunto de clases e interfaces programadas en el propio Java.

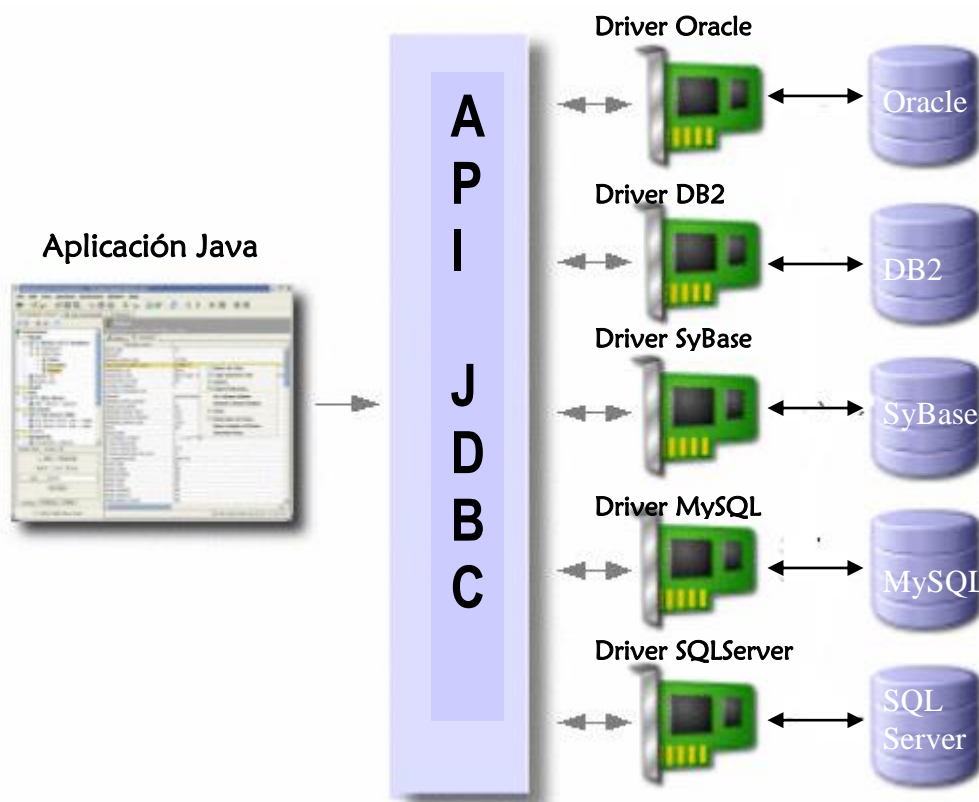
Drivers JDBC

Para acceder a BBDD en Java es necesario:

1. El entorno de desarrollo de Java (JDK)
2. La API JDBC (incluida en el JDK)
3. **Disponer del driver Java específico para el gestor de BBDD escogido**

Para usar JDBC con un sistema gestor de base de datos en particular, es necesario disponer del driver JDBC apropiado que haga de intermediario entre ésta y JDBC.

Un driver JDBC es el programa que permite a la aplicación Java interactuar con la base de datos. En concreto, es una clase que implementa la interfaz JDBC Driver, proporciona la conexión al SGBD y convierte solicitudes SQL para una BD en particular



¿Qué haremos con JDBC?

- Establecer una conexión con la base de datos.
- Enviar sentencias SQL, de selección y actualización.
- Procesar los resultados.

El siguiente fragmento de código nos muestra un ejemplo básico de estas tres cosas:

```
Connection con = DriverManager.getConnection (
    " jdbc:mysql://localhost:3306/bd ", "login", password");
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT a, b, c FROM Tabla1");
while (rs.next()) {
    int x = rs.getInt("a");
    String s = rs.getString("b");
    float f = rs.getFloat("c");
}
rs.close();
st.close();
con.close();
```

Clases principales

Las clases JDBC se incluyen en el paquete **java.sql**, siendo éstas las más utilizadas.

TIPO	Clase JDBC
Implementación	java.sql.Driver java.sql.DriverManager java.sql.DriverPropertyInfo
Conexión a base de datos	java.sql.Connection
Sentencias SQL	java.sql.Statement java.sql.PreparedStatement java.sql.CallableStatement
Datos	java.sql.ResultSet
Errores	java.sql.SQLException java.sql.SQLWarning

SQLException

La mayoría de los métodos del API JDBC pueden lanzar la excepción **SQLException**

- Esta excepción se produce cuando se da algún error en el acceso a la base de datos: error de conexión, sentencias SQL incorrectas,...
- Es un tipo de excepción “caught” (que hay que tratar)

```
try{ .....
}
catch(SQLException e){
    //Interesante utilizar: e.getMessage() con el error concreto generado por el SGBD.
    //Por ejemplo: “ORA-00942: la tabla o vista no existe”
}
```

Clase DriverManager

Registro del driver JDBC

Para poder conectarse a una BD, es preciso antes cargar el driver encargado de esta función.

Class.forName(String driver)

Ejemplos

`Class.forName("com.mysql.jdbc.Driver");` ← Registro de driver para MySQL

`Class.forName("oracle.jdbc.driver.OracleDriver");` ← Registro de driver para Oracle

** tendréis que incluir en el proyecto la librería (.jar) correspondiente*

Establecimiento de una conexión

El método `getConnection` establece una conexión a la base de datos cuya url se pasa como parámetro, normalmente aportando unas credenciales.

static Connection getConnection (String url, String user, String password)

`jdbc:mysql://<host>:<puerto>/< base de datos>` ← url de conexión a mysql

`jdbc:oracle:thin:@<host>:<puerto>:<base de datos>` ← url de conexión a Oracle

Ejemplos

```
Connection con=DriverManager.getConnection
("jdbc:mysql://localhost:3306/bdciclos", "Juan","1234");
```

```
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:orcl", "c##scott", "tiger");
```

Clase Connection

Un objeto de la clase `Connection` representa una sesión conectados con una base de datos. Se establece, como acabamos de ver, mediante el método `getConnection` de `DriverManager`.

Creación de un `Statement`

La función principal de la clase `Connection` es crear objetos tipo `Statement`, mediante el método `createStatement`

`Statement createStatement()` Crea un objeto que permitirá enviar consultas `SQL` a la `BD`. Puede haber varios `Statement` simultáneos sobre la misma conexión.

Ejemplo:

```
Statement st=con.createStatement( );
Statement st2=con.createStatement( );
```

`void close()` Libera la conexión con la base de datos y los recursos `JDBC`

Clase Statement

Un `Statement` representa el canal por el que se envían instrucciones `SQL` a la base de datos y se reciben los resultados. Vale para instrucciones `DML` (incluido `SELECT`) y `DDL`

`ResultSet executeQuery(String sql)` Ejecuta una sentencia `SQL SELECT` que devuelve un objeto `ResultSet` (conjuntos de tuplas).

`int executeUpdate(String sql)` Ejecuta una sentencia `INSERT`, `UPDATE`, `DELETE`, o una sentencia `DDL` (cualquier sentencia `SQL` que no devuelva nada)
Devuelve 0 o el `Nº` de filas afectadas por la actualización

`void close()`

Ejemplos:

```
Connection con = DriverManager.getConnection
    ("jdbc:oracle:thin:@localhost:1521:orcl", "c##scott", "tiger");
Statement st=con.createStatement( );
ResultSet productos = st.executeQuery("SELECT * FROM productos WHERE precio<10");
st.executeUpdate("INSERT INTO cafes VALUES ('Colombiano',9)");
st.close();
```

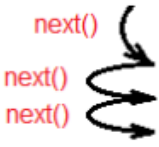
Clase ResultSet

Contiene los datos resultado (filas) de una sentencia SQL Select.

Simula una tabla con las filas y columnas devueltas.

Existen métodos para recuperar secuencialmente las filas de un ResultSet, mediante un cursor que se desplaza automáticamente

```
Select Emp_Id, Emp_No, Emp_Name from Empl
```



	1	2	3
	EMP_ID	EMP_NO	EMP_NAME
1	7839	E7839	KING
2	7566	E7566	JONES
3	7902	E7902	FORD
4	7369	E7369	SMITH
5	7698	E7698	BLAKE
6	7499	E7499	ALLEN

boolean **next()**

Avanza a la siguiente fila, desde la posición actual. En un principio, el cursor está situado ANTES de la primera fila del ResultSet, por lo que será necesario una primera llamada a next() para acceder a ella. Devuelve **false** cuando el cursor se sale de la hoja de resultados

** **IMPORTANTE:** incluso cuando sepamos que el resultset consta de una sola fila/registro, es necesario hacer next() para acceder a ella y recuperar valores*

int **getInt** (int columna)
int **getInt** (String nombre_columna)

Métodos que permiten inspeccionar el 'ResultSet': Devuelven, para la fila actual, el valor de una columna a partir de su nombre o de su posición (ojo! **comenzando en 1**, no es un índice),

float **getFloat** (int columna)
float **getFloat** (String nombre_columna)

String **getString** (int columna)
String **getString** (String nombre_columna)

El método getString permite leer cualquier campo de la tabla (aunque sea numérico) como cadena de caracteres.

java.sql.Date **getDate**(int columna)
java.sql.Date **getDate**(String nombre_colum)
etc

Hay que tener en cuenta que getDate recupera campos fecha con el tipo java.sql.Date.

Correspondencia de
tipos SQL ↔ método getXXX

SQL Type	Java Method
BIGINT	getLong()
BINARY	getBytes()
BIT	getBoolean()
CHAR	getString()
DATE	getDate()
DECIMAL	getBigDecimal()
DOUBLE	getDouble()
FLOAT	getDouble()
INTEGER	getInt()
LONGVARBINARY	getBytes()
LONGVARCHAR	getString()
NUMERIC	getBigDecimal()
OTHER	getObject()
REAL	getFloat()
SMALLINT	getShort()
TIME	getTime()
TIMESTAMP	getTimestamp()
TINYINT	getByte()
VARBINARY	getBytes()
VARCHAR	getString()

void **close()**

Ejemplos

Dadas la tabla de clientes y compras de una BD MySQL **bdcompras**:

	id	nombre	pais	edad	vip
+	1	Joao	Portugal	44	<input type="checkbox"/>
+	2	Boris	Croacia	34	<input type="checkbox"/>
+	3	Pierre	Canadá	49	<input type="checkbox"/>
+	4	Julio	Portugal	24	<input type="checkbox"/>
+	5	Pavel	Croacia	2	<input type="checkbox"/>

	idcompra	idcliente	concepto	importe
	1	3	AB209	100
	2	1	X9222	50
	3	3	AC300	150
	4	4	D299	300
	5	5	D900	45
	6	2	D299	300
	7	2	W255	20
	8	3	L73	120
	9	5	AC300	400

Ejemplo

Visualizar nombre y edad de clientes cuya edad es mayor de la media

```
try {
    Class.forName("com.mysql.jdbc.Driver" );
    String url = " jdbc:mysql://localhost:3306/bdcompras ";
    Connection con = DriverManager.getConnection(url, "Juan", "1234");

    String sql="select nombre, edad from clientes where edad >(select avg(edad) from clientes)";
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery(sql);

    while(rs.next()){
        S.O.P("Nombre:"+ rs.getString(1));
        S.O.P(",edad: "+ rs.getInt("edad"));
    }
    rs.close();
    st.close();
    con.close();

} catch (ClassNotFoundException e) {
    System.out.println("Error driver");
} catch (SQLException e) {
    System.out.println(e.toString());
}
```

Visualiza

Nombre:Joao,edad: 44
 Nombre:Boris,edad: 34
 Nombre:Pierre,edad: 49

Configurar como clientes vip (actualizar campo vip ← true) aquellos clientes cuyas compras sumen más de 400 euros

```
Class.forName("com.mysql.jdbc.Driver" );
String url = " jdbc:mysql://localhost:3306/bdcompras ";
Connection con = DriverManager.getConnection(url, "Juan", "1234");
String sql="UPDATE clientes SET vip = true WHERE id in
(select idcliente from compras group by idcliente having sum(importe)>400)";
Statement st = con.createStatement( );
int n=st.executeUpdate(sql);
System.out.println(n + " clientes actualizados");
st.close();
con.close();
```

clientes : Tabla						
		id	nombre	pais	edad	vip
▶	+	1	Joao	Portugal	44	<input type="checkbox"/>
	+	2	Boris	Croacia	34	<input type="checkbox"/>
	+	3	Pierre	Canadá	49	<input checked="" type="checkbox"/>
	+	4	Julio	Portugal	24	<input type="checkbox"/>
	+	5	Pavel	Croacia	2	<input checked="" type="checkbox"/>

Clase **PreparedStatement** (Derivada de Statement)

Representa objetos que contienen órdenes SQL precompiladas

Se utiliza cuando se quiere ejecutar varias veces la misma consulta con distintos valores. Al estar precompilada, la ejecución será más rápida.

La consulta SQL contenida en un objeto **PreparedStatement** puede tener **uno o más parámetros** cuyo valor no se especifica cuando se crea. Dichos parámetros se especifican mediante interrogantes: ?

Cada interrogante se identifica posteriormente con un número.

Finalmente, al ejecutar la consulta, rellenaremos esos valores con los métodos **setXXX()**

Los métodos setXXX tienen 2 parámetros:

- El primer parámetro es el número del interrogante, de forma que el primer interrogante que aparece es el 1, el segundo el 2, etc.
- El segundo es el valor a insertar en el lugar del interrogante

En general, hay un método setXXX para cada tipo Java (setInt, setFloat, setString, etc)

Ejemplo 1

```
Connection con = DriverManager.getConnection(.....);
```

```
List<String> lista=new ArrayList<String>();
```

```
PreparedStatement ps = con.prepareStatement("SELECT * FROM Clientes WHERE pais = ?");
```

```
ps.setString(1, txtPais.getText());
```

```
ResultSet rs = ps.executeQuery();
```

```
while (rs.next()) {
    lista.addItem(rs.getString("Titulo"));
}
rs.close();
ps.close();
con.close();
```

Ejemplo 2

```
Cliente[] clientes={
    new Cliente(10,"Jon",40}, new Cliente(11,"Ana",35}, ....new Cliente(12,"Jim",60));
.....
```

```
Connection con = DriverManager.getConnection (.....);
```

```
String sql ="INSERT INTO clientes (id,nombre, edad) VALUES( ?, ?, ?)";
```

```
PreparedStatement ps = con.prepareStatement (sql); ; → Se precompila la consulta
```

```
for (int i=0; i<clientes.length; i++){
    ps.setString (1,clientes[i].getId());
    ps.setString (2, clientes[i].getNombre());
    ps.setInt (3, clientes[i].getEdad());
    int n = ps.executeUpdate( );
```

→ Cada ejecución de la consulta es más rápida,
al estar precompilada}

```
}
ps.close();
con.close();
```


Ejemplo 3: Insertar nuevas filas en la tabla de compras anterior (extraídas de un fichero de texto)

Compras : Tabla				
	idcompra	idcliente	concepto	importe
	1	3	AB209	100
	2	1	X9222	50
	3	3	AC300	150
	4	4	D299	300

Fichero de texto **compras.txt**

(En una línea el tabulador separa los datos)

10	4	PK400	27.5
11	2	D299	300
12	1	KL9001	70.09
...

```
Connection con = DriverManager.getConnection(.....);
```

```
PreparedStatement ps = con.prepareStatement("INSERT INTO Compras "+
"(idcompra,idcliente, concepto, importe) VALUES ( ?, ?, ?, ?)");
```

```
BufferedReader br=new BufferedReader(new FileReader("compras.txt"));
```

```
String str=br.readLine();
```

```
while(str!=null ){
```

```
    String[] partes = str.split("\t");
```

```
    ps.setString(1, partes[0]);
```

```
    ps.setString(2, partes[1]);
```

```
    ps.setString(3, partes[2]);
```

```
    ps.setFloat(4, Float.parseFloat(partes[3]));
```

```
    ps.executeUpdate();
```

```
    str=br.readLine( );
```

```
}
```

```
br.close();
```

```
ps.close();
```

```
con.close();
```

Utilizar un pool de conexiones en JavaEE

Si preveemos que nuestra aplicación web tendrá cientos o miles de usuarios concurrentes, es necesario preocuparse por el rendimiento del servidor.

- En este sentido, la conexión/desconexión a bases de datos es una tarea que consume muchos recursos (CPU, Memoria)
- Además, si no se cierran las conexiones aun teniendo pocos usuarios es probable que acabemos saturando al servidor de conexiones sin usar.

Por ello, conviene gestionar los accesos a las BBDD mediante un pool de conexiones.

Un **Pool de Conexiones** es un conjunto de conexiones ya establecidas a una base de datos (gestionado por el servidor de aplicaciones), pudiendo ser estas conexiones reutilizadas por las diferentes peticiones.

Tener un Pool de Conexiones permite centralizar el acceso a la base de datos, ya que de dicho acceso se encarga el servidor de aplicaciones.

La idea de usar un Pool de Conexiones es que cada vez que un cliente necesita una conexión a la base de datos, la solicita al Pool.

- Dicha conexión se asigna al cliente hasta que no la necesita más, y en ese momento, en lugar de cerrarla, se devuelve al pool, para poder ser aprovechada por otro cliente.
- Si un cliente necesita una conexión y no hay ninguna disponible, se queda esperando hasta que alguna conexión del pool se libere.

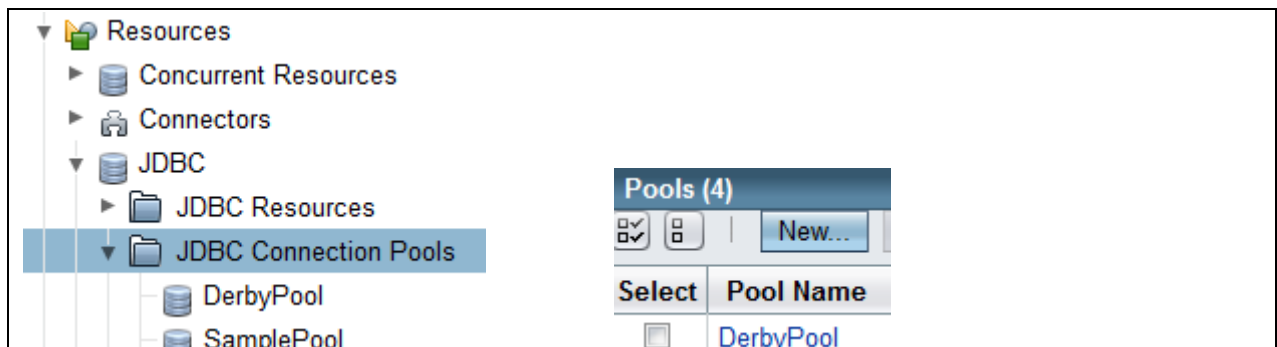
Las conexiones en un Pool de Conexiones se mantienen abiertas desde el principio, por lo que los clientes se ahorran el tiempo de conexión a la Base de Datos.

(Una petición web clásica –sin pool- que acceda a una BD se conecta, consulta o actualiza y se desconecta. Dicha conexión y desconexión consume recursos valiosos para el servidor)

Creación de un pool de conexiones en GlassFish

Hay muchas maneras de crear un pool de conexiones, dependiendo qué servidor y qué SGBD usemos. Veremos cómo crear un pool de conexiones JDBC en GlassFish.

1. Entrar a la consola de Admon de Glassfish: <http://localhost:4848> (con GlassFish en marcha, claro)
 ** En la versión 4.1.1 de Glassfish (el q viene bundled con Netbeans 8.1,8.2) hay un error que impide crear recursos JDBC desde la consola visual
2. Entrar a “JDBC Connection Pools” y añadir un nuevo pool en “New”



3. Darle 1 nombre, de tipo `javax.sql.DataSource` y Driver: **MySql**

General Settings

Pool Name: *

Resource Type:

Must be specified if the datasource class imp

Database Driver Vendor:

Select or enter a database driver vendor


Introspect: ☒ **Enabled**

If enabled, data source or driver implementati

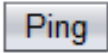
4. Meter las propiedades **servidor**, **nombrebd**, **puerto** (que suele ser 3306 para mysql), **usuario** y **password**, (El resto las eliminamos)

Select	Name	Value
<input type="checkbox"/>	password	123456
<input type="checkbox"/>	databaseName	bdbus
<input type="checkbox"/>	serverName	localhost
<input type="checkbox"/>	user	root
<input type="checkbox"/>	portNumber	3306

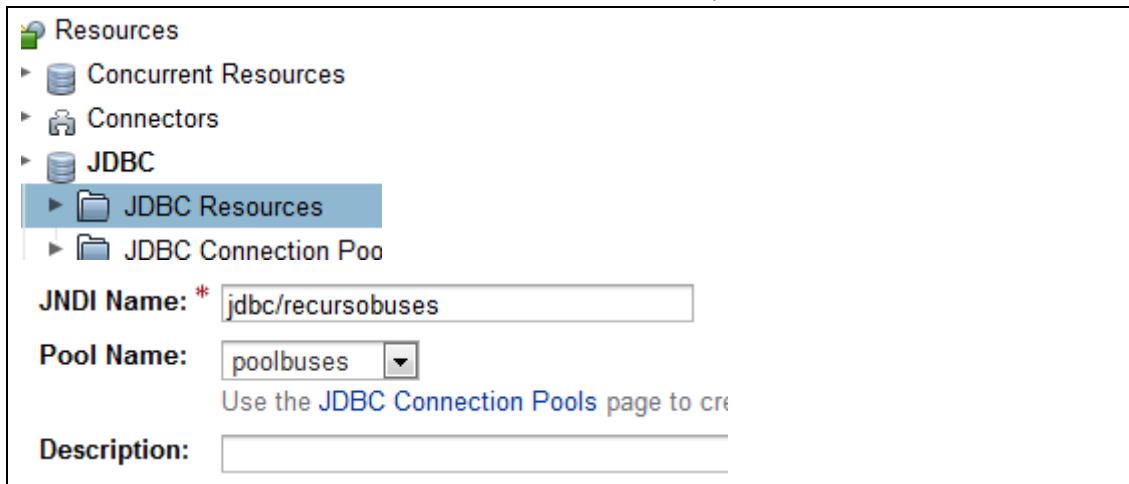
5. GlassFish necesita q le copiemos “el jar de MySQL” en su carpeta de librerías.

La carpeta de GlassFish/lib debe contener, pues,  `mysql-connector-java-5.1.23-bin.jar`
Se lo podríamos copiar, por ejemplo, de la carpeta de NetBeans

6. Hacer ping para comprobar que se conecta:



7. Crear un recurso JDBC con un nombre JNDI, asociado al pool anteriormente creado



En nuestra clase Java

Desde Java podemos acceder al pool de conexiones usando la interfaz *javax.sql.DataSource*. Al llamar al método *getConnection()* se obtendrá una conexión del pool de conexiones y no se creará nueva en ese momento.

Para obtener un objeto DataSource, simplemente debemos saber que un mismo servidor web puede tener muchos pool de conexiones, por lo que identifica cada uno de ellos mediante un nombre JNDI.

```
// Uso del poolContext
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/nombre_recurso_jndi");

//Obtener la conexión del Pool
Connection cn = ds.getConnection();
```

Cuando llamemos al método *java.sql.Connection.close()* realmente la conexión no se cerrará sino que se devolverá al pool de conexiones al que pertenece.

```
//Devolver la conexión al pool
cn.close();
```

De este modo, nuestros proyectos JavaEE pueden abrir y cerrar conexiones tantas veces como lo requieran (una por cada consulta a la BD) sin que esto repercute en el rendimiento de la aplicación.

Establecer pools de conexiones a nivel de programación (desde Java)

La librería de apache **commons-dbc** provee una implementación sencilla de un pool de conexiones, para cuando no tengamos la opción de configurarla desde el servidor.

Esta librería necesita, a su vez, las librerías **commons-pool** y **commons-login**.

Su clase **BasicDataSource** crea, internamente, un pool de conexiones, por lo que bastará con indicar en un lugar las características del BasicDataSource e invocar el método (p.e getConnection) para tomar una conexión del pool, cada vez que necesitemos ejecutar una consulta.

Ejemplo de uso

```
public final class PoolConex {
    private static final BasicDataSource dataSource = new BasicDataSource();

    static {
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost/bd");
        dataSource.setUsername("usuario");
        dataSource.setPassword("password");

        //OPCIONAL (si no, se usarán datos por defecto para el pool)
        // dataSource.setMinIdle(5);
        // dataSource.setMaxIdle(20);
        // dataSource.setMaxOpenPreparedStatements(180);
        // dataSource.setMaxTotal(10);
    }

    public static Connection getConnection() throws SQLException {
        return dataSource.getConnection();
    }
}

public class Dao {
    public static Usuario login(String user, String password) {
        Usuario user=null;
        try (Connection con=PoolConex.getConnection();
            String sql = "select .....from .... where id=" + user + " and pass="+password+"";
            Statement st = con.createStatement();
            ResultSet rs = st.executeQuery(sql);
            if(rs.next())
                user=new Usuario(rs.getString("..."),rs.getString("..."));
            rs.close();
            st.close();
        )
        catch (Exception e) {
            System.err.println("Error en login: " + e);
        }
        return user;
    }
}
```

```
try ( ) { } catch { }
```

es un ejemplo de try-with-resources.

Todos los recursos (cn, st,...) que se abran en el paréntesis se cerrarán automáticamente (close implícito) en orden inverso al de apertura

TRATAMIENTO DE FECHAS

Nos manejaremos con 2 tipos de fechas:

- java.util.Date, para fechas genéricas (las fechas de un bean)
- java.sql.Date, para las fechas recuperadas de una BD

, y por otra parte están las representaciones textuales de las fechas (Simple Strings tipo “2020/07/03”)

A continuación se muestra cómo realizar las conversiones entre estos tipos:

PARTIENDO DE UNA FECHA java.util.Date.....

java.util.Date fecha=new java.util.Date(); → fecha actual

java.util.Date fecha=objeto.getFecha(); → propiedad fecha de un bean

→ Convertirla a texto

```
SimpleDateFormat formateador = new SimpleDateFormat("dd/MM/yyyy");  
String fechastr= formateador.format(fecha);
```

Ejemplo: para una consulta sql donde la fecha va como texto

```
String sqlinsert="insert into tabla(....., campofecha) values(.....fechastr)";
```

PARTIENDO DE UNA FECHA SQL RECUPERADA DE UNA BD java.sql.Date

```
java.sql.Date fechasql=rs.getDate("campofecha");
```

→ Convertirla a fecha java.util.Date:

```
java.util.Date fecha = new java.util.Date(fechasql.getTime());
```

Ejemplo: para asignar una fecha de la BD a un atributo de un bean

```
objeto.setFecha(new java.util.Date(rs.getDate("campofecha").getTime()));
```

Hay que tener en cuenta que rs.getDate("campo_datetime") devuelve la información de fecha sin hora. Cuando estemos interesados en la hora, una solución es la siguiente:

```
java.util.Date fecha=new java.util.Date(rs.getTimestamp("campo").getTime());
```

PARTIENDO DE UN STRING tipo “2020/05/25”

```
String fechastr="2020/08/25";
```

→ Convertirla a fecha java.util.Date:

```
SimpleDateFormat formateador = new SimpleDateFormat("yyyy/MM/dd");  
java.util.Date fecha =formateador.parse(fechastr);
```

Ejemplo: recoger fecha de un JTextfield y asignarla a atributo de un bean

```
String strFecha=txtFecha.getText();  
Objeto.setFecha (new SimpleDateFormat("yyyy/MM/dd").parse(strFecha));
```