GESTIÓN DE BASES DE DATOS RELACIONALES

Contenido

1 Introducción

Lo primero que se necesita para conectarnos desde una aplicación con una base de datos es un **driver** o **Conector**. Ese driver es el elemento que, de alguna forma, sabe cómo hablar con la base de datos.

Según el gestor de base de datos que se utilice, se usará un driver u otro.

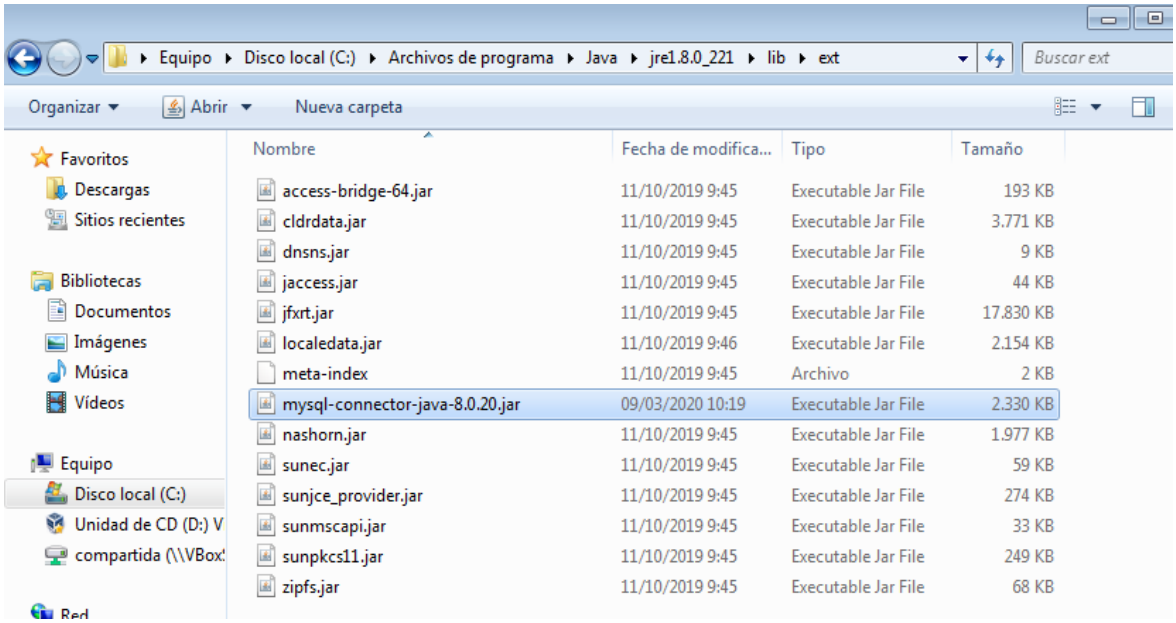
Para **MySql** se usa ***mysql-connector-java-X.X.X-bin.jar.***

Para **Oracle** se usa ***ojdbc6*.jar**.

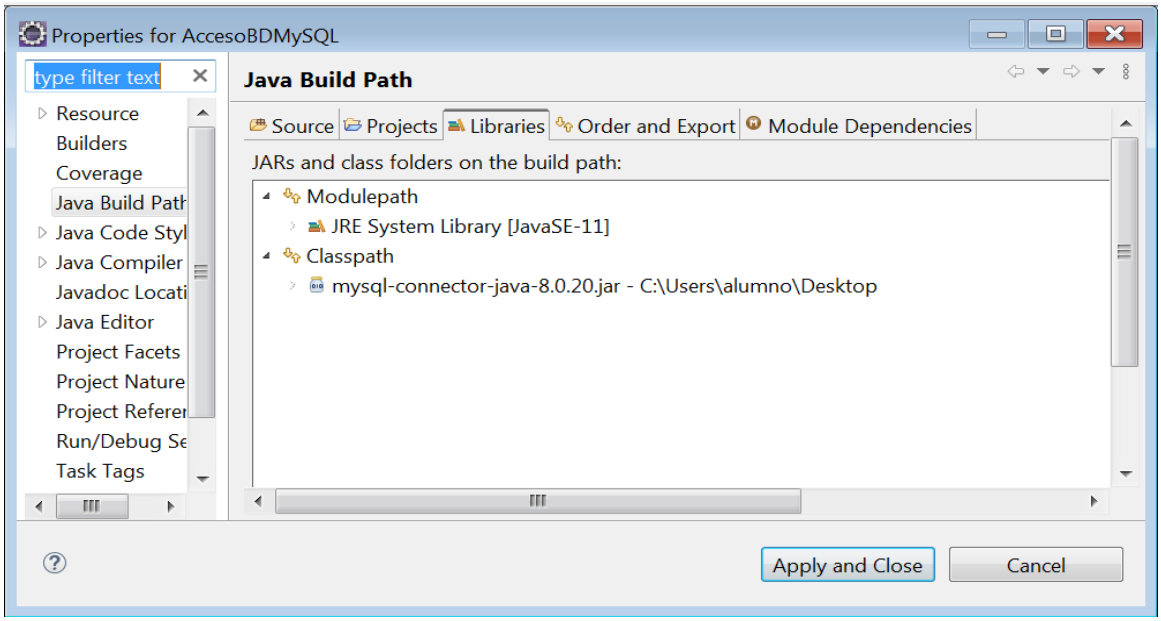
Para poder conectar MySQL con Java vamos a utilizar la tecnología JDBC de Java. JDBC nos permite **el acceso a los datos desde un programa Java**.

El fichero que contiene el driver mysql-connector-java-X.X.X.jar se encontrará dentro del *classpath* de la aplicación, ya que iremos a dicho *classpath* para cargarlo y utilizarlo en nuestro programa.

Podemos colocar el driver en cualquier sitio, por ejemplo en la carpeta del jre/jdk.



O podemos dejarlo sencillamente en el escritorio y hacer que nuestro proyecto tenga esa localización en el classpath.



Cargar el Driver MySQL

- Si estás utilizando una **versión anterior a JDBC 4**, es decir, si utilizas un compilador **inferior a JavaSE 7**, lo primero que necesitarás será cargar el driver. Para cargar el driver acudiremos a la clase Class.forName(), la cual recibirá como parámetro el nombre del driver.

  ```
  Class.forName("com.mysql.jdbc.Driver").newInstance();
  ```

- Si estás utilizando **Java SE 7 o superior** puedes evitar esta línea de código ya que el compilador cargará el Driver automáticamente por nosotros.

Conexión a MySQL desde JDBC

Una vez tengamos el Driver cargado deberemos conectarnos a la base de datos mediante la clase **Connection** y su método **.getConnection()**. Dicho método espera una **cadena de conexión** a la base de datos. La cadena de conexión a MySQL tiene la siguiente estructura:

jdbc:mysql://[host1]:[port1],[host2]:[port2]/[databaseName]?[property1=value1]&[property2=value2]

Vamos a instalar la aplicación XAMPP, que nos permitirá usar la herramienta phpMyAdmin y la versión propia de MySQL.

En resumen, nuestro trabajo en este tema va a consistir en llevar a cabo los siguientes pasos:

Los **pasos** a seguir para trabajar con la base de datos desde Java son:

1º.- Importar el paquete java.sql.*
2º.- Cargar el driver para lo cual se pone Class.forName("Nombre del driver") si es necesario.
Para el caso del MySql, será: Class.forName("com.mysql.jdbc.Driver");

3º.- Conectarse a la base de datos invocando el método getConnection().

La **sintaxis** de **getConnection** es:

Connection conn = DriverManager.getConnection(url, usr, pwd);

Donde **url** es una cadena compuesta por el protocolo + driver + lugar donde se encuentra la base de datos + nombre de la bbdd, **usr** es el usuario para acceder a la bbdd y **pwd** es la clave de acceso.

Para el caso de MySql sería (si no hemos puesto ninguna contraseña):

url: "jdbc:mysql://localhost/Nombre_de_la_Base_de _Datos"

usuario:"root"

password:""

Todas estas instrucciones estarán recogidas dentro de un try-catch para que recoja la excepción en caso de no poder realizarse la conexión, tener un error de SQL o cualquier otra excepción.

Ejemplo:

```
try{
        Class.forName("com.mysql.jdbc.Driver");
        Connection conn=
        DriverManager.getConnection("jdbc:mysql://localhost/bbdd","root","");
        ……..
}catch(ClassNotFoundException cnfe){
        System.out.println("Driver JBDC no encontrado");
}catch(SQLException sqle){
        System.out.println("Error al conectarse a la BD");
}catch(Exception e){
        System.out.println("Error general");
```

4º.- Una vez creada la conexión se crean las sentencias SQL, a través de los métodos execute(), executeUpdate() y executeQuery() de la interface Statement.

Statement st=conn.createStatement():

El método **execute**() se utiliza para las operaciones **DDL** sobre tablas (Create, Alter o Drop).
El método **executeUpdate**() se utiliza para operaciones **DML**, sobre los registros (Update, Insert o Delete). Devuelve un número entero que indica la cantidad de registros afectados.
El método **executeQuery**() se utiliza para las consultas **SELECT**. Devuelve un conjunto de registros que se almacenan en un objeto **ResultSet**.

Ejemplo **execute**:

```
st.execute("CREATE TABLE Libros (id NUMBER(11) primary key,
title VARCHAR2(64))");
```

Ejemplo **executeUpdate**:

```
int nr;
String cnsSQL;
cnsSQL="INSERT INTO autor VALUES(127,'Peter','Norton')";
nr=st.executeUpdate(cnsSQL);
cnsSQL="UPDATE autor SET nombre='Pedro' WHERE
nombre='Peter'";
nr=st. executeUpdate(cnsSQL);
```

Ejemplo **executeQuery**:

```
ResultSet rs=st.executeQuery("SELECT * FROM autor");
```

ResultSet dispone de los métodos **next()** y **getXXX()** que nos permitirán acceder a los datos que nos ha proporcionado el servidor de base de datos:

next() recorre los registros devueltos.
getXXX() recupera los valores de las columnas por su nombre o posición.

Ejemplo:

```
while(rs.next()){
        System.out.println("Autor "+rs.getString(2));
}
```

Cuando se lanza un método **getXXX** sobre un objeto **ResultSet**, el driver JDBC convierte el dato que se quiere recuperar al tipo Java especificado y entonces devuelve un valor Java adecuado. La conversión de tipos se puede realizar gracias a la clase java.sql.Types. En esta clase se definen lo que se denominan tipos de datos JDBC, que se corresponde con los tipos de datos SQL.

5º.- Por último habrá que cerrar todos los Statement y las conexiones con el método **close()**.

```
st.close();

conn.close();
```

Correspondencia datos Java-SQL:

SQL	Java
BIGINT	getLong()
BINARY	getBytes()
BIT	getBoolean()
CHAR	getString()
DATE	getDate()
DECIMAL	getBigDecimal()
DOUBLE	getDouble()
FLOAT	getDouble()
INTEGER	getInt()
LONGVARBINARY	getBytes()

SQL	Java
LONGVARCHAR	getString()
NUMERIC	getBigDecimal()
OTHER	getObject()
REAL	getFloat()
SMALLINT	getShort()
TIME	getTime()
TIMESTAMP	getTimestamp()
TINYINT	getByte()
VARBINARY	getBytes()
VARCHAR	getString()

Ejemplo:

```
import java.sql.*;
public class Programa {
public static void main(String args[]){
try {
Class.forName("com.mysql.jdbc.Driver");
Connection conexion;
conexion=DriverManager.getConnection
("jdbc:mysql://localhost/ejemplo","root","");
Statement instruccion = conexion.createStatement();
```

```
ResultSet tabla;
tabla= instruccion.executeQuery("SELECT cod , nombre
FROM datos");
System.out.println("Codigo\tNombre");
while(tabla.next())
                    System.out.println(tabla.getInt(1)+"\t"+tabla.getString(
                    2));
tabla.close();
instruccion.close();
conexion.close();
}catch(ClassNotFoundException e) {
System.out.println(e);
}catch(SQLException e) {
System.out.println(e);
}catch(Exception e) {
System.out.println(e);
            }
      }
}
```

Probamos a usar el **try con recursos** para que se cierre automáticamente la base de datos.

CONSIDERACIONES

- Si la sentencia SQL ejecutada con el método executeQuery() no devolviera un conjunto de registros resultado (p.ej. INSERT INTO...), obtendríamos una excepción SQLException.
- Por defecto, un objeto Statement tiene solamente un ResultSet abierto. Por tanto, si queremos tener más de un ResultSet abierto simultáneamente, deberemos utilizar distintos objetos Statement's.

Existe un objeto **ResultSetMetaData** que proporciona varios métodos para obtener información sobre los datos que están dentro de un objeto ResultSet. Estos métodos permiten entre otras cosas obtener de manera dinámica el número de columnas en el conjunto de resultados, así como el nombre y el tipo de cada columna.

NOTA (solo si es necesario):

En alguna versión del driver de mySQL, da un error si realizamos la conexión como se ha indicado antes. Debemos añadir la zona horaria.

```
conexion=DriverManager.getConnection
("jdbc:mysql://localhost/Prueba?useUnicode=true&use"+

"JDBCCompliantTimezoneShift&useLegacyDatetimeCode=false&ServerTimezone=UTC",

"root","");
```

2 La interfaz PreparedStatement

La interfaz **PreparedStatement** hereda de **Statement** y difiere de esta de dos maneras:

- Las instancias de **PreparedStatement** contienen una sentencia SQL que ya ha sido compilada. Esto es lo que hace que se le llame 'preparada'.
- La sentencia SQL contenida en un objeto **PreparedStatement** puede tener uno o más parámetros IN. Un parámetro IN es aquel cuyo valor no se especifica en la sentencia SQL cuando se crea. En vez de ello, la sentencia tiene un interrogante ('?') como un 'ancla' para cada parámetro IN. Debemos suministrar un valor para cada interrogante mediante el método apropiado, que puede ser: **setInt**, **setString**, etc, antes de ejecutar la sentencia.

Un objeto **PreparedStatement** se usa para sentencias SQL que toman uno o más parámetros como argumentos de entrada (parámetros IN).

Conviene que nos aseguremos de que el servidor está capacitado para trabajar con sentencias precompiladas, lo hacemos de la siguiente forma:

```
Connection conexion =
        DriverManager.getConnection(
            "jdbc:mysql://servidor/basedatos?useServerPrepStmts=true",
            "usuario", "password");
```

PreparedStatement tiene un **grupo de métodos** que fijan los valores de los parámetros IN, los cuales son enviados a la base de datos cuando se procesa la sentencia SQL.

Las Instancias de PreparedStatement extienden, es decir, heredan de Statement y por tanto heredan los métodos de Statement.

Un **objeto PreparedStatement** es potencialmente más eficiente que un objeto Statement porque este ha sido precompilado y almacenado para su uso futuro.

Son muy útiles cuando una sentencia SQL se ejecuta muchas veces cambiando solo algunos valores.

- Se utiliza para enviar al servidor sentencias SQL precompiladas con uno o más parámetros.
- Se crea un objeto PreparedStatement especificando la plantilla y los lugares donde irán los parámetros.
- Los parámetros son especificados después utilizando los métodos **setXXX**(.) indicando el número de parámetro y el dato a insertar en la sentencia.
- La sentencia SQL y los parámetros se envían al gestor base de datos cuando se llama al método: executeXXX()

Los objetos PreparedStatement contienen órdenes SQL **precompiladas** que pueden por tanto ejecutarse muchas veces.

Ejemplos:

El siguiente fragmento de código, donde **con** es un **objeto Connection**, crea un objeto **PreparedStatement** que contiene una instrucción SQL:

```
// Creamos un objeto PreparedStatement desde el objeto Connection

PreparedStatement ps = con.prepareStatement(
”select * from Propietarios where DNI=? AND NOMBRE=? AND EDAD=?”);

// ”Seteamos” los datos al prepared statement de la siguiente forma (los segundos parámetros son variables con valores):


ps.setString(1, dni);
ps.setString(2, nombre);
ps.setInt(3, edad);

//Ejecutamos el PreparedStatement, en este caso con executeQuery():


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

Ejemplo de PreparedStatement de consulta:

Por ejemplo, supongamos que hay un campo de texto en el que el usuario puede introducir su dirección de correo electrónico y con este dato se desea buscar al usuario:

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

String consulta = "SELECT usuario FROM registro WHERE email like ?";
PreparedStatement pstmt = con.prepareStatement(consulta);
pstmt.setString(1 , campoTexto.getText());

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

Ejemplo de PreparedStatement de modificación:

En el siguiente ejemplo se va a insertar un nuevo registro en una tabla:

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

String insercion = "INSERT INTO registro(usuario , email , fechaNac) values ( ? , ? , ? )");
PreparedStatement pstmt = con.prepareStatement(insercion);

String user = . . . ;
String email = . . . ;
Date edad = . . . ; //O int edad;
pstmt.setString(1, user);
pstmt.setString(2, email);
pstmt.setDate(3, edad);  // setInt(3, edad);
pstmt.executeUpdate();
```

Como los objetos **PreparedStatement** están precompilados, su ejecución es más rápida que los objetos **Statement**. Consecuentemente, una sentencia SQL que se ejecute muchas veces, a menudo se crea como **PreparedStatement** para incrementar su eficacia.

Siendo una subclase de **Statement**, **PreparedStatement**, como ya se ha mencionado hereda toda la funcionalidad de **Statement**. Además, se añade un conjunto completo de métodos necesarios para fijar los valores que van a ser enviados a la base de datos en el lugar de las 'anclas' para los parámetros IN.

También se modifican los tres métodos **execute**, **executeQuery** y

executeUpdate de tal forma que ***no toman argumentos***. Los formatos de **Statement** de estos métodos (los formatos que toman una sentencia SQL como argumento) no deberían usarse nunca con objetos **PreparedStatement**, a pesar de estar disponibles para ellos.

Los métodos de PreparedStatement

Ejecución de órdenes: executeQuery()

public abstract ResultSet executeQuery() throws SQLException

Devuelve la tabla resultado de ejecutar la orden precompilada. No devuelve nunca un valor null.

Ejecución de órdenes de actualización: executeUpdate()

public abstract int executeUpdate() throws SQLException

Devuelve el número de filas afectadas por órdenes del tipo INSERT, UPDATE, DELETE, o sentencias de definición de datos.

Parámetros nulos: setNull()

public abstract void setNull(int *indiceParametro*, int *tipoSQL*) throws SQLException

Asigna un valor nulo a un parámetro. No puede omitirse el tipo SQL.

Asignación de valores a los parámetros: setXXX()

public abstract void setXXX(int *indiceParametro*, *tipoJava valor*) throws SQLException

Asigna el valor valor del tipo Java tipoJava al parámetro de índice indiceParametro, que es transformado por el controlador JDBC en un tipo SQL correspondiente para pasarlo a la base de datos.

Método setXXX()	*Tipo Java*	*Tipo SQL*
setBoolean	boolean	BIT
setByte	byte	TINYINT
setShort	short	SMALLINT
setInt	int	INTEGER
setLong	long	BIGINT
setFloat	float	FLOAT

setDouble	double	DOUBLE
setBigInt	BigInt	NUMERIC
setString	String	VARCHAR o LONGVARCHAR
setBytes	Array de bytes	VARBINARY o LONGVARBINARY
setDate	Date	DATE
setTime	Time	TIME
setTimestamp	Timestamp	TIMESTAMP
setAsciiStream*	InputStream	LONGVARCHAR
setUnicodeStream*	InputStream	LONGVARCHAR
setByteStream*	InputStream	LONGVARBINARY

public abstract void setXXX(int *indiceParametro*, inputStream *corriente* , int *longitud*) throws SQLException

Se usan para valores muy grandes ASCII, UNICODE o binarios.

Borrado de parámetros: clearParameters()

public abstract void clearParameters() throws SQLException

Borra todos los parámetros**.**

Asignación de objetos: setObject()

void setObject(int *indiceParametro*, Object *objeto* [, int *tipoSQL*][, int *decimales*]) throws SQLException

Asigna un objeto a un parámetro para ser convertido al tipo SQL especificado. Si el tipo SQL no se especifica, se emplean las tablas de conversión del estándar JDBC para entregarlo a la base de datos como el tipo adecuado.

Ordenes con múltiples resultados: execute()

public abstract boolean execute() throws SQLException

Se emplea para enviar órdenes que pueden devolver múltiples resultados.

Nota: El índice de los parámetros comienza en 1.

http://tutorials.jenkov.com/jdbc/preparedstatement.html