Bases de Datos con JAVA

Juan Carlos Pérez Rodríguez

Sumario

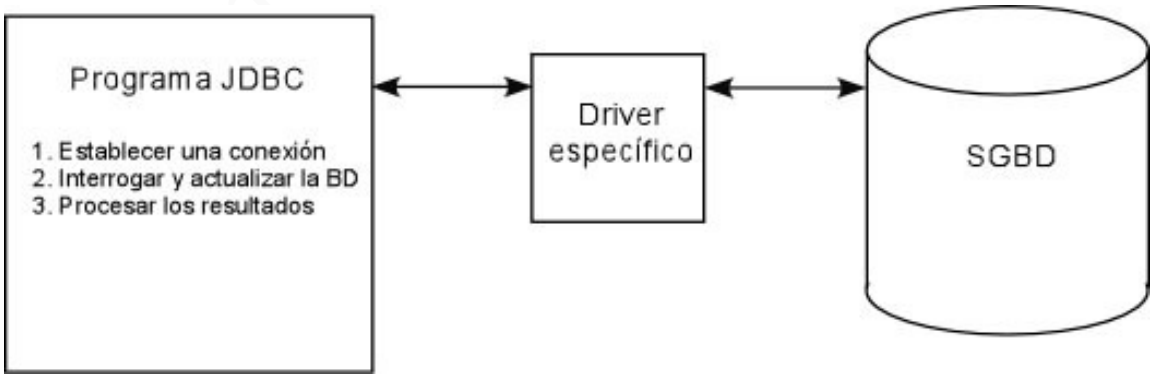
Introducción

Las bases de datos será nuestra forma habitual de mantener la información de nuestra aplicación. Son sistemas optimizados y por tanto los más eficientes para gestionar la persistencia de nuestra aplicación.

Dejaremos la parte de persistencia que corresponde a los ficheros a casos muy específicos (por ejemplo, en un editor de texto. Las preferencias de usuario de nuestra aplicación etc) Pero ¿ cómo nos conectaremos a la base de datos ?

JDBC: Java Database Connectivity (en español: Conectividad a bases de datos de Java), más conocida por sus siglas JDBC, es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto SQL del modelo de base de datos que se utilice. (fuente wikipedia)

JDBC nos permitirá acceder a bases de datos (BD) desde Java. Con JDBC no es necesario escribir distintos programas para distintas BD, sino que un único programa sirve para acceder a BD de distinta naturaleza. Incluso, podemos acceder a más de una BD de distinta fuente (Oracle, Access, MySql, etc.) en la misma aplicación. Podemos pensar en JDBC como el puente entre una base de datos y nuestro programa Java.



La impedancia objeto-relacional

El desfase objeto-relacional, también conocido como impedancia objeto-relacional, consiste en la diferencia de aspectos que existen entre la programación orientada a objetos y la base de datos. Estos aspectos se puede presentar en cuestiones como:

Lenguaje de programación. El programador debe conocer el lenguaje de programación orientada a objetos (POO) y el lenguaje de acceso a datos.

Tipos de datos: en las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, que suelen ser sencillos, mientras que la programación orientada a objetos utiliza tipos de datos más complejos.

Paradigma de programación. En el proceso de diseño y construcción del software se tiene que hacer una traducción del modelo orientado a objetos de clases al modelo Entidad-Relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas o filas, lo que implica que se tengan que diseñar dos diagramas diferentes para el diseño de la aplicación.

El modelo relacional trata con relaciones y conjuntos debido a su naturaleza matemática. Sin embargo, el modelo de Programación Orientada a Objetos trata con objetos y las asociaciones entre ellos. Por esta razón, el problema entre estos dos modelos surge en el momento de querer persistir los objetos de negocio

La escritura (y de manera similar la lectura) mediante JDBC implica: abrir una conexión, crear una sentencia en SQL y copiar todos los valores de las propiedades de un objeto en la sentencia, ejecutarla y así almacenar el objeto. Esto es sencillo para un caso simple, pero trabajoso si el objeto posee muchas propiedades, o bien se necesita almacenar un objeto que a su vez posee una colección de otros elementos. Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.

Este problema es lo que denominábamos impedancia Objeto-Relacional, o sea, el conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito en con u lenguajes de Programación Orientada a Objetos.

Podemos poner como ejemplo de desfase objeto-relacional, un Equipo de fútbol, que tenga un atributo que sea una colección de objetos de la clase Jugador. Cada jugador tiene un atributo "teléfono". Al transformar éste caso a relacional se ocuparía más de una tabla para almacenar la información, implicando varias sentencias SQL y bastante código.

Procedimiento para conectar mediante JDBC

Para usar JDBC hay que seguir los siguientes pasos:

1.- Incluir el jar con el driver del SGBD. Para esto se puede ir a la página de la empresa (por ejemplo mysql da la posibilidad de descargar el driver jdbc) y luego incluirlo entre las librerías. Otra alternativa (mejor) es usar maven o gradle para gestionar las dependencias

Si lo hacemos con maven podemos buscar en google (nos dará un enlace a maven central) maven mysql-connector-java

Una vez en maven central tomamos la versión que queramos del driver. Por ejemplo veamos la 8.0.20:

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.20</version>
</dependency>
```

2.- Registrar el driver. Para que se pueda usar el driver tenemos que “cargarlo”

El siguiente método se puede usar para cargar un driver mysql:

```
    private static  void cargarDriverMysql(){
      try {
        Class.forName("com.mysql.cj.jdbc.Driver");
      } catch(ClassNotFoundException ex) {
        System.err.println("no carga el driver");
        System.exit(1);
      }
    }
```

3.- Establecer la conexión.

Los controladores se identifican con un URL JDBC de la forma:

```
jdbc:subprotocolo:localizadorBD
```

• El subprotocolo indica el tipo de base de datos específico

• El localizador permite referenciar de forma única una BD

  • Host y opcionalmente puerto

  • Nombre de la base de datos

La conexión a la BD se hace con el método getConnection()

- public static Connection getConnection(String url)

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

- public static Connection getConnection(String url, Properties info)

Tener en cuenta que todos pueden lanzar la excepción SQLException

Ejemplo:

```
try {
  conexion = DriverManager.getConnection(
     "jdbc:mysql://localhost/tienda","pruebas", "clavepruebas");
} catch (SQLException ex) {
// Tratar el error
}
```

En el código anterior le decimos que el subprotocolo(tipo de DB específica) es: mysql. En la parte de localizador aparece: **localhost** que indica que es en la propia máquina donde está la base de datos. Y la parte que dice: **tienda** especifica que la base de datos se llama: tienda. Finalmente

nos aparece el nombre del usuario que se conecta: **pruebas** y la password para ese usuario: **clavepruebas**

4.- Crear y ejecutar operaciones en la DB

Prepararemos la sentencia a ejecutar (createStatement() o prepareStatement()) la lanzamos (excuteUpdate(), executeQuery(), execute(),…) Para finalmente tomar los resultados devueltos (ResultSet) y cerraremos la conexión

Ejemplo:

Veamos la base de datos:

```
create database oficina;
use oficina;
SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
START TRANSACTION;
SET time_zone = "+00:00";
CREATE TABLE `lapices` (
  `idlapiz` int NOT NULL,
  `marca` char(30) NOT NULL,
  `numero` int DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

INSERT INTO `lapices` (`idlapiz`, `marca`, `numero`) VALUES
(1, 'staedtler', 2),
(2, 'alpino', 1),
(3, 'alpino', 3),
(4, 'staedtler', 1);
ALTER TABLE `lapices`
  ADD PRIMARY KEY (`idlapiz`);
COMMIT;
```

Una primera versión podría ser:

```
public class Main {
    private static  void cargarDriverMysql(){
      try {
        Class.forName("com.mysql.cj.jdbc.Driver");
      } catch(ClassNotFoundException ex) {
        System.err.println("no carga el driver");
        System.exit(1);
      }
    }

    public static void main(String[] args) {

        cargarDriverMysql();
        Connection conexion = null;
        try {
            conexion = DriverManager.getConnection(
                    "jdbc:mysql://localhost/oficina?serverTimezone=UTC","root",
"1q2w3e4r");

            Statement s = conexion.createStatement();
            String sql = "select * from lapices";
            ResultSet rs = s.executeQuery(sql);

            while(rs.next()){

                int id = rs.getInt("idlapiz");
                String marca = rs.getString("marca");
                int numero = rs.getInt("numero");

                System.out.println("Lápiz: " + id + " " + marca + " " + numero);

            }
            s.close();
            conexion.close();

        } catch (SQLException ex) { ex.printStackTrace(); }
    }
}
```

El código anterior primero registra el driver y luego intenta hacer una conexión:

conexion = DriverManager.getConnection(

"jdbc:mysql://localhost/oficina?serverTimezone=UTC","root", "1q2w3e4r")

la uri nos indica que nos conectamos a mysql, a la base de datos: oficina con el usuario: root y la password: 1q2w3e4r Todo eso lo hemos comentado antes. Pero ahora observamos el parámetro: serverTimezone=UTC

Bien, este parámetro tiene lugar porque las versiones del driver desde hace algún tiempo precisan conocer de la zona horaria. Esto lo hacen para evitar incoherencia en la lectura de los datos. Imaginar una base de datos centralizada y una aplicación en otro servidor que querrá mostrar a sus clientes la información según su zona horaria (no la hora que está guardada en la base de datos que será diferente probablemente de la del server de la aplicación) En nuestro caso siempre agregaremos el servertimezone con UTC establecido

Veamos las siguiente 3 líneas:

```
Statement s = conexion.createStatement();
String sql = "select * from lapices";
ResultSet rs = s.executeQuery(sql);
```

Hay varias formas de ejecutar una consulta contra la DDBB. Una es mediante: createStatement() Básicamente estamos lanzando una consulta en sql plano (sin compilar ni parametrizar) mediante el método executeQuery() y tomamos los resultados En un conjunto: ResultSet

ResultSet contiene todas las filas que satisfacen la consulta sql lanzada. Para ir avanzando al siguiente registro usamos: ResultSet.next()

¡**¡importante!!** resultset no está apuntando al primer registro una vez obtenido mediante executeQuery(). Para ubicarnos en el primer registro tenemos que ejecutar ResultSet.next()

Veamos ahora el recorrido de los registros:

```
while(rs.next()){
    int id = rs.getInt("idlapiz");
    String marca = rs.getString("marca");
    int numero = rs.getInt("numero");
```

```
                System.out.println("Lápiz: " + id + " " + marca + " " + numero);
            }
            s.close();
```

Observar que con rs.next() estamos avanzando de un registro a otro. Una vez ubicados en un registro, tomamos la información de cada campo mediante métodos como: rs.getInt(), rs.getString(),..

Los métodos rs.getInt(), rs.getString() reciben un nombre de campo en String o un entero que hace referencia al número de columna que corresponde a un campo en concreto.

Así en el ejemplo anterior sería coincidente:

```
String marca = rs.getString("marca");
String marca = rs.getString(2);
```

En ambas sentencias obtenemos el campo marca del registro, ya que marca es la segunda columna de la query lanzada

Finalmente cerramos la statement y después la conexión

Práctica 1: Crear la aplicación anterior en la que el usuario introduzca el nombre de la marca y se le muestren los registros correspondientes a la marca solicitada. Debe acceder convenientemente a la base de datos y hacer uso de una clase java: Lapiz que permita reconstruir un arraylist de lapices desde el resultset obtenido. Hacer una versión de texto y otra gráfica. Observar que podemos obtener todos los lapices en el resultset y únicamente poner en el arraylist los de la marca solicitada. También podemos ejecutar la query con una cláusula where

Práctica 2: Crear una versión de la aplicación anterior en la que tengamos una clase llamada: GestorLapices con un constructor que reciba el nombre de la base de datos (en el ejemplo que hemos realizado el nombre era: oficina) y que tenga un método llamado: obtenerLapicesPorMarca() que se le pase el nombre de la marca y devuelva un arraylist de lapices

En las actividades anteriores se ha nombrado que se puede lanzar la query a la DDBB con un where respecto a marca o también filtrar los datos recibidos en Java ¿ qué supone hacerlo en Java ?

Sabemos que las bases de datos son eficientes con las consultas y también sabemos que si nuestra base de datos está en red si solicitamos más datos de los que precisamos aumenta el tráfico innecesariamente. Normalmente evitaremos hacer el filtrado de datos en Java si podemos pedírselo directamente a la base de datos

Ahora estudiaremos como lanzar las diferentes consultas:

- Statement

- PreparedStatement

- CallableStatement

SQL Injection

“Inyección SQL es un método de infiltración de código intruso .

El origen de la vulnerabilidad radica en la incorrecta comprobación o filtrado de las variables utilizadas en un programa que contiene, o bien genera, código SQL.

Se dice que existe o se produjo una inyección SQL cuando, de alguna manera, se inserta o "inyecta" código SQL invasor dentro del código SQL programado, a fin de alterar el funcionamiento normal del programa y lograr así que se ejecute la porción de código "invasor" incrustado, en la base de datos”

Si observamos el ejemplo anterior, vemos que el desarrollador estaba solicitando el contenido de una **variable** (fechadenacimiento) Que es justo lo que se indica en la definición de wikipedia:

```
UPDATE alumnos SET fechanacimiento="
```

Y el atacante en lugar de darle únicamente un valor, agrega, “inyecta” código SQL (observar el concepto de que es código, no un valor, porque ese es el matiz para tratar de controlar la vulnerabilidad como veremos en la siguiente sección)

Para lidiar con este tipo de ataques tenemos opciones como las **PreparedStatement** que vemos en la siguiente sección

PreparedStatement

Lo primero y más importante que debemos entender es que estas sentencias admiten que les pasemos parámetros (observar el interrogante: “?” que es el lugar donde va el parámetro)

```
"SELECT * FROM asignaturas WHERE idasignatura =  ? "
```

Estas sentencias pasan de esta forma (literalmente) a nuestro SGBD . Allí se compila la sentencia. Después se inserta el valor (únicamente admite valores para la parte de interrogantes, no admite nada más) que se le pase como parámetro, y finalmente ejecuta

De lo anterior podemos concluir dos ventajas: La principal que podemos combatir mejor contra el sql injection ya que lo que envíe el usuario y que sea permitido serán únicamente valores que se reemplazarán allí donde haya interrogantes. La segunda es que al quedar compilada en el SGBD puede recurrirse a ella nuevamente para ejecuciones posteriores más rápidas

Veamos un ejemplo:

```
String query = "INSERT INTO asignaturas(nombre,curso) VALUES(?,?) " ;

// cn es de tipo Connection

PreparedStatement ps;
ps = cn.prepareStatement(query,PreparedStatement.RETURN_GENERATED_KEYS);

ps.setString(1, "BAE");
ps.setString(2, "1º DAM");

ps.executeUpdate();
```

Observar que le pasamos los parámetros. El primer interrogante que aparezca será el parámetro 1, el segundo interrogante será el parámetro 2 etc.

Así en el código anterior, BAE se está poniendo como nombre y 1ºDAM como curso

Por lo demás es muy parecido a Statement, existen executeUpdate, executeQuery,…

Cuidado al hacer uso de: executeQuery(), executeUpdate() porque hay dos versiones: una recibe una String (que realmente no es propio de la clase PreparedStatement sino de su padre: Statement) y la otra versión no lleva String, que es la que realmente queremos usar.

El siguiente código es correcto:

```
String query = "SELECT * FROM asignaturas WHERE idasignatura =  ? " ;
PreparedStatement ps = cn.prepareStatement(query);
ps.setInt(1, (int)id);

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

Pero si en lugar de la última línea hubiéramos puesto:

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

entonces habría fallado. Ya que se desencadenaría la versión del método executeQuery() perteneciente a la clase Statement en lugar de la de PreparedStatement

Un ejemplo que refleja como buscar un registro en la tabla asignaturas:

```
int id = 1;
String query = "SELECT * FROM asignaturas WHERE idasignatura =  ? " ;
try (
      Connection cn = gc.getConnection();
    PreparedStatement ps = cn.prepareStatement(query);
 ){
  ps.setInt(1, id);

      ResultSet rs = ps.executeQuery();
      if(rs.next()) {
            asignatura = new Asignatura();
            asignatura.setIdasignatura(id);
            asignatura.setNombre(rs.getString("nombre"));
            asignatura.setCurso(rs.getString("curso"));
      }
} catch (SQLException e) { e.printStackTrace();}
```

Agregar nulos en PreparedStatement

Para establecer un nulo podemos emplear setNull() Debemos conocer el tipo de dato para especificarlo con Types. Supongamos que en la base de datos corresponde un entero y que es el parámetro 3 el que queremos poner a nulo, entonces:

```
ps.setNull(3,java.sql.Types.INTEGER);
```

Procedimientos Almacenados

Un procedimiento almacenado es un procedimiento o subprograma que está almacenado en la base de datos.

Un procedimiento almacenado MySQL no es más que una porción de código que puedes guardar y reutilizar. Es útil cuando repites la misma tarea repetidas veces, siendo un buen método para encapsular el código. Al igual que ocurre con las funciones, también puede aceptar datos como parámetros, de modo que actúa en base a éstos.

Muchos sistemas gestores de bases de datos los soportan, por ejemplo: MySQL, Oracle, etc.

Tipos:

- Procedimientos almacenados.
- Funciones, las cuales devuelven un valor que se puede emplear en otras sentencias SQL.

Al reducir la carga en las capas superiores de al aplicación, se reduce el tráfico de red y, si el uso de los procedimientos almacenados es el correcto, puede mejorar el rendimiento.

Al encapsular las operaciones en el propio código SQL, nos aseguramos de que el acceso a los datos es consistente entre todas las aplicaciones que hagan uso de ellos.

En cuanto a seguridad, es posible limitar los permisos de acceso de usuario a los procedimientos almacenados y no a las tablas directamente. Des este modo evitamos problemas derivados de una aplicación mal programada que haga un mal uso de las tablas.

Un procedimiento almacenado típico tiene:

Un nombre.

Una lista de parámetros.

Unas sentencias SQL.

Veamos un ejemplo de sentencia para crear un procedimiento almacenado sencillo para MySQL, aunque sería similar en otros sistemas gestores:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS alumnossubnombre$$
CREATE PROCEDURE alumnossubnombre(IN subnombre VARCHAR(50))
BEGIN
  SELECT *
  FROM alumnos
  WHERE alumnos.nombre LIKE  subnombre;
END
$$
```

Observar que el delimitador de sentencias habitual es: “;” Se cambia para que el sistema entienda que es una única sentencia (la creación de un procedimiento almacenado) ya que dentro las acciones que se realizan pueden ser varias y precisar de finalización. En ese caso el punto y coma que se ponga no lo interpretará como final del procedimiento almacenados

En el anterior ejemplo se crea un procedimiento que devuelve el listado de productos para una variable que toma de entrada. La forma de llamar al procedimiento es:

```
Connection cn = gc.getConnection();
try {

      CallableStatement pc = cn.prepareCall("call alumnossubnombre(?)");
      pc.setString(1,"%a%");
      ResultSet rs = pc.executeQuery();
      while(rs.next()) {
            System.out.println(rs.getString("nombre"));
      }

} catch (SQLException e) {e.printStackTrace(); }
```

La consulta anterior devuelve todos los alumnos que tienen una letra a en su nombre. Observar que hemos puesto el tanto por ciento: **“%”** como un parámetro. Este es el comportamiento habitual que haremos. NO pondremos dentro de la query el: “%” en las prepareStatement ni en las prepareCall sino que se lo pasaremos al enviar el parámetro

DAO

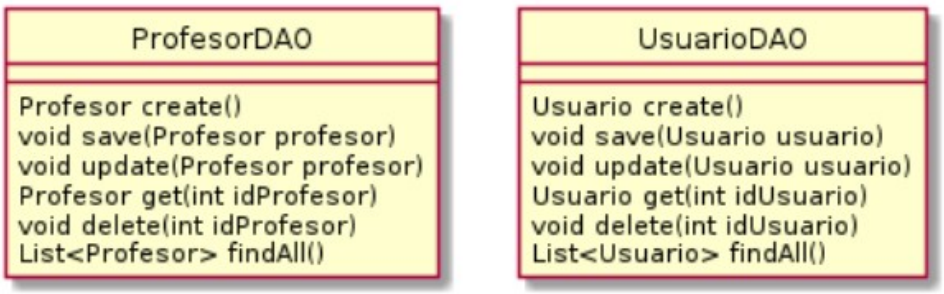
Según wikipedia:

En software de computadores, un objeto de acceso a datos (en inglés, data access object, abreviado DAO) es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de datos o un archivo.

Los Objetos de Acceso a Datos son un Patrón de los subordinados de Diseño Core J2EE y considerados una buena práctica. La ventaja de usar objetos de acceso a datos es que cualquier objeto de negocio (aquel que contiene detalles específicos de operación o aplicación) no requiere conocimiento directo del destino final de la información que manipula.

Los Objetos de Acceso a Datos pueden usarse en Java para aislar a una aplicación de la tecnología de persistencia Java subyacente (API de Persistencia Java), la cual podría ser JDBC, JDO, Enterprise JavaBeans, TopLink, EclipseLink, Hibernate, iBATIS, o cualquier otra tecnología de persistencia

El modelo DAO podría incluso tomar por cada clase que necesitamos crear una clase equivalente DAO. Por ejemplo, supongamos que nuestra aplicación usará dos clases que queremos persistir. Una se llama Profesor y la otra Usuario entonces el patrón DAO nos podría dar:



Si nos fijamos, entonces con la línea:

```
profesorDAO.update( profesor )
```

estaríamos en ese método update() tocando la base de datos para actualizar la información del objeto: profesor que tenemos en memoria

Se propone pues, crear un objeto DAO por cada una de las tablas/objetos necesarias. En nuestra aplicación precisaríamos como mínimo:

AlumnoDAO, AsignaturaDAO, MatriculaDAO

Se propone que TODOS ellos implementen el siguiente interfaz:

```
public interface Crud<T,E> {

	T save(T dao);

	T findById(E id);

	boolean update(T dao);

	boolean delete(E id);

	ArrayList<T> findAll();
}
```

Donde el tipo genérico: T hace referencia a la clase que queremos trabajar (por ejemplo en AlumnoDAO sería Alumno) y el tipo: E hace referencia al ID que usamos para esa clase (el ID de un Alumno es un dni que es de tipo String)

Veamos como quedaría el prototipo de AlumnoDAO entonces:

```
public class AlumnoDAO implements Crud<Alumno,String>{

	GestorConexionDDBB gc;
	public AlumnoDAO(GestorConexionDDBB gc) {
		this.gc = gc;
	}

	@Override
	public Alumno save(Alumno dao) {
		String query="INSERT INTO alumnos(dni,nombre ...";
		try (
			Connection cn = gc.getConnection();
		    PreparedStatement ps = cn.prepareStatement(query);
		 ){
			//aquí el código que corresponda
		}
		return null;
	}

	@Override
	public Alumno findById(String id) {
		return null;
	}

	@Override
	public boolean update(Alumno dao) {
		return false;
	}

	@Override
	public boolean delete(String id) {
		return false;
	}

	@Override
	public ArrayList<Alumno> findAll() {
		return null;
	}
}
```

Observar que todos los override son exactamente lo que viene del interfaz

Por otro lado aparece un constructor que recibe `GestorConexionDDBB` eso es debido a que hemos creado una clase que maneja la conexión. Vamos a verla

```
public class GestorConexionDDBB {

    String jdbcUrl;
    String usuario;
    String clave;

    public Connection getConnection() {

        Connection con=null;
        try {
            con = DriverManager.getConnection(jdbcUrl, usuario, clave);
        } catch (SQLException ex) {
            System.exit(1);
        }
        return con;
    }

    public GestorConexionDDBB(String ddbb,String nombreUsuario, String password)
{
        jdbcUrl = "jdbc:mysql://localhost/"+ddbb+"?serverTimezone=UTC";
        usuario = nombreUsuario;
        clave = password;
        cargarDriverMysql();
    }

    private static  void cargarDriverMysql(){
      try {
        Class.forName("com.mysql.cj.jdbc.Driver");
      } catch(ClassNotFoundException ex) {
        System.err.println("no carga el driver");
        System.exit(1);
      }
    }
}
```

Como vemos, lo único que hace es establecer y fijarnos todo para que lo único que tengamos que hacer cuando queremos una nueva conexión sea: GestorConexionDDBB.getConnection()

Transacciones

Pongámonos en el siguiente caso de nuestra aplicación. Tenemos que guardar una nueva matrícula. Como dijimos ya, parece lógico que en el momento de la matrícula se establezcan cuáles son las asignaturas que se van a cursar. Eso significa que tenemos que hacer un INSERT en la tabla matriculas y tantos INSERT como asignaturas se vayan a cursar en la tabla: asignatura_matricula

¿ Qué ocurre si insertamos correctamente el registro en la tabla matrícula y sin embargo falla la inserción en la tabla asignatura_matricula ? Si lo dejáramos así sería que el alumno estará matriculado pero no tiene asignaturas que cursar. Eso no tiene sentido. Debiéramos entender toda la acción como una única y atómica, de tal forma que si no se consiguen insertar todos los registros que corresponde para dar de alta la matrícula, entonces que no se tenga en cuenta ninguno

Pues precisamente eso es lo que es una **transacción**: Agrupa un conjunto de operaciones como una única operación. Si alguna de ellas falla entonces se deshacen las otras

Las bases de datos que soportan transacciones son mucho más seguras y fáciles de recuperar si se produce algún fallo en el servidor que almacena la base de datos, ya que las consultas se ejecutan o no en su totalidad.

Al ejecutar una transacción, el motor de base de datos garantiza: atomicidad, consistencia, aislamiento y durabilidad (ACID) de la transacción (o conjunto de comandos) que se utilice.

El ejemplo típico que se pone para hacer más clara la necesidad de transacciones en algunos casos es el de una transacción bancaria. Por ejemplo, si una cantidad de dinero es transferida de la cuenta de un cliente, pongamos: clienteEmisor a otro cliente, pongamos: clienteReceptor, hace falta dos operaciones en DDBB:

```
UPDATE cuentas SET saldo = saldo - cantidad WHERE cliente = "clienteEmisor";

UPDATE cuentas SET saldo = saldo + cantidad WHERE cliente = "clienteReceptor";
```

Si la segunda operación no se pudiera realizar pero la primera sí ocurriría que el dinero "desaparecería" ya que habría sido quitado de la cuenta del emisor pero no agregado a la del receptor

Commit, Rollback

Una transacción tiene dos finales posibles, COMMIT o ROLLBACK. Si se finaliza correctamente y sin problemas se hará con COMMIT, con lo que los cambios se realizan en la base de datos, y si por alguna razón hay un fallo, se deshacen los cambios efectuados hasta ese momento, con la ejecución de ROLLBACK.

Habitualmente en los SGBD, e**n una conexión trabajamos en modo autocommit** c**on valor true**. Eso significa que cada consulta es una transacción en la base de datos.

Por tanto, si queremos definir una transacción de varias operaciones, estableceremos el modo autocommit a false con el método setAutoCommit de la clase Connection.

En modo **no autocommit** las transacciones quedan definidas por las ejecuciones de los métodos commit y rollback. Una transacción abarca desde el último commit o rollback hasta el siguiente commit.

Ej:

```
BEGIN

...

SET AUTOCOMMIT OFF
update cuenta set saldo=saldo + 250 where dni="12345678-L";
update cuenta set saldo=saldo - 250 where dni="89009999-L";
COMMIT;
...

EXCEPTION
   WHEN OTHERS THEN
      ROLLBACK ;
END;
```

La anterior transacción es código de un procedimiento anónimo que se ejecuta DENTRO DE LA BASE DE DATOS. Así que ahora sabemos que las transacciones son nativas. Veamos como enviamos las instrucciones desde Java con el ejercicio de matrículas:

Cuando modificamos una Matricula, implica también a las asignaturas de la matrícula en la tabla asignatura_matricula

Así se necesitan estas 3 query:

```
String queryDelete = "DELETE FROM asignatura_matricula  WHERE idmatricula =  ? "
;
String queryUpdate = "UPDATE matriculas SET dni= ?, year= ?  WHERE idmatricula =
? " ;


String queryInsert = "INSERT INTO asignatura_matricula(idasignatura,idmatricula)
VALUES(?,?) " ;
```

Primero borramos (delete) toda asociación de las asignaturas con la matrícula (las filas de asignatura_matricula son la clave aquí)

Después modificamos (update) la tabla matriculas para poner los posibles cambios introducidos (el dni del alumno, el año de la matricula)

finalmente agregamos (insert) en la tabla asignatura_matricula las nuevas asociaciones que nos hayan dado

Lo lógico es que **ninguna modificación tenga lugar si alguna de las sentencias tenga lugar.** Así que **hay que cubrir con una transacción**

Veamos un trozo del código necesario:

```
String queryDelete = "DELETE FROM asignatura_matricula  WHERE idmatricula =  ? " ;
String queryUpdate = "UPDATE matriculas SET dni= ?, year= ?  WHERE idmatricula =  ? " ;


String queryInsert = "INSERT INTO asignatura_matricula(idasignatura,idmatricula)
VALUES(?,?) " ;
try (
      Connection cn = gc.getConnection();

      PreparedStatement psDelete = cn.prepareStatement(queryDelete);
      PreparedStatement psUpdate = cn.prepareStatement(queryUpdate);
      PreparedStatement psInsert = cn.prepareStatement(queryInsert);
){
      cn.setAutoCommit(false);

      psDelete.setInt(1, dao.getIdmatricula());
      int respuesta = psDelete.executeUpdate();

      psUpdate.setString(1, dao.getAlumno().getDni());
      psUpdate.setInt(2, dao.getYear());
      psUpdate.setInt(3, dao.getIdmatricula());


      respuesta = psUpdate.executeUpdate();

      ok = respuesta > 0;  //entendemos que debe haber alguna fila borrada si ok

      if(ok ) {

            //… .. .. ←Aquí el resto del código
            //si todo sale bien, al final hay un commit
            cn.commit();
            cn.setAutoCommit(true);

       }else{   //aquí salió mal así que “rompemos la transacción”
          cn.rollback();
          cn.setAutoCommit(true);
       }
}
```

Vemos que hemos usado un try-with-resources en el código anterior. Y es que estamos siempre expuestos al: SQLException

Excepciones.

Las conexiones con una base de datos consumen muchos recursos en el sistema gestor, y por lo tanto en el sistema informático en general. Por ello, conviene cerrarlas con el método `close` siempre que vayan a dejar de ser utilizadas, en lugar de esperar a que el garbage collector de Java las elimine.

También conviene cerrar las consultas (`Statement` y `PreparedStatement`) y los resultados (`ResultSet`) para liberar los recursos.

Una excepción es una situación que no se puede resolver y que provoca la detención del programa de manera abrupta. Se produce por una condición de error en tiempo de ejecución.

En base de datos es especialmente interesante el uso de try with resourses porque el proceso de manejo y cierre de excepciones puede ser un poco tedioso

Pongamos como ejemplo una situación que se da en la aplicación de matrículas. En el código que vimos antes de modificación de una matrícula hay que abrir y ejecutar múltiples preparedStatement y todas ellas pueden ser susceptibles de lanzar excepción. Habría que cubrirse con un try y finalmente cerrar todo correctamente. Con try-with-resources se nos facilita bastante ya que tienen implementado el autocierre.