Acceso a BBDD SQL

Un proyecto de acceso a BBDD SQL se compone de la siguiente arquitectura:

```
principal.java
model
    |__PrimerObjecto.java
    |__SegundoObjeto.java
    |__TercerObjeto.java
dao
    |__PrimerObjetoDAO.java
    |__SegundoObjetoDAO.java
    |__TercerObjetoDAO.java
database
    |__DBScheme.java
    |__DBConnection.java
```

database -> DBScheme.java

Aquí incluiremos las constantes que nos permitirán acceder a la base de datos. De esta forma no tendremos que estar recodando como se llaman, directamente accederemos a los valores guardados aquí.

```java
package database;

public interface DBScheme {

    // Almacen de constantes variables -> finales (no de metodos)
    String DB_NAME = "nombre_base_datos";
    String HOST = "127.0.0.1";
    String PORT = "3306";

    // credenciales
    String USER = "root";
    String PASSWORD = "";

    // Tablas
    String TAB_COCHE = "coches";
```

```
    String TAB_PAS = "pasajeros";
    String TAB_COCHE_PAS = "coches_pasajeros";

    // Columnas del coche
    String COL_ID = "id";
    String COL_COCHE_MATRICULA= "matricula";
    String COL_COCHE_MARCA= "marca";

    // Columnas del pasajero
    String COL_PAS_NOMBRE = "nombre";
    String COL_PAS_EDAD = "edad";
    String COL_PAS_PESO = "peso";

}
```

database -> DBConnection.java

Esta clase es la que se usará para asegurar la conexión. Siempre tendrá la misma estructura, así que no te preocupes en modificarlo. Es copiar y pegar.

```
package database;
import java.sql.*;
public class DBConnection {

    private static Connection connection;

    // Obtenemos la conexión
    public Connection getConnection() {
        if (this.connection == null) {
            try {
                System.out.println("Estableciendo conexión");
                createConnection();
            } catch (SQLException e) {
                System.out.println("Error al establecer conexión");
                throw new RuntimeException(e);
            }
        }
        return this.connection;
    }

    // Creamos la conexión
    public void createConnection() throws SQLException {
        String url = String.format("jdbc:mysql://%s:%s/%s", DBScheme.HOST,
DBScheme.PORT, DBScheme.DATABASE);
        this.connection = DriverManager.getConnection(url, DBScheme.USER,
DBScheme.PASSWORD);
        System.out.println("Conexión establecida correctamente.");
        listDatabases();
    }

    // Cerramos la conexión
```

```
    public void closeConnection() {
        try {
            this.connection.close();
            this.connection = null;
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    // Método para listar las bases de datos disponibles
    private void listDatabases() throws SQLException {
        String query = "SHOW DATABASES";
        PreparedStatement preparedStatement = connection.prepareStatement(query);
        ResultSet resultSet =  preparedStatement.executeQuery();
        System.out.println("Bases de datos disponibles:");
        while (resultSet.next()){
            System.out.println("- " + resultSet.getString(1));
        }

    }

}
```

Ahora que tenemos la conexión, podemos realizar peticiones desde cualquier clase del proyecto. Normalmente, las peticiones las meteremos dentro de cada DAO, tal y como veremos aquí:

Constructor y declaración de propiedades

Declaramos las propiedades y ya. Como es un DAO, normalmente no habrá un constructor, sino que podemos declarar la conexión desde dentro. Como observación, iniciaremos la conexión desde dentro de cada método con un new Connection(), aunque no es la práctica más eficiente, servirá para este ejercicio.

```
public class CochesDAO {

    private Connection connection;
    private PreparedStatement preparedStatement;
    private ResultSet resultSet;

    // A partir de aquí, podemos meter los distintos métodos

    addOne( Coche coche ){}

    findById( int id ){}

    findAll(){}

    deleteOne( int id ){}

    replaceOne( Coche coche ){}
```

```
}
```

Sintaxis de una consulta

La mejor forma de realizar las consultae con StringFormat + PreparedStatement. PreparedStatement es la forma segura de hacer las consultas, mientras que StringFormat nos proporciona una forma más legible de hacer las consultas.

StringFormat

- %s -> String. Representa un string que en los parametros sucesivos será remplazado, en este caso por el nombre de la tabla o columna, que obtendremos de la base de datos.
- ? -> representa el valor que se le asignará a la columna. Para asignar un valor a ? posteriormente deberemos utilizar los métodos setString, setInt, etc.

PreparedStatement Para preparar el statemente seguiremos tres pasos:

1. Crear la conexión y llamar al método prepareStatement(query).
2. Asignar los valores a los ? mediante los métodos setString, setInt, etc.
3. Ejecutar la consulta mediante executeQuery() o executeUpdate().

Opcionalmente, podemos retornar el valor de la consulta o almacenarlo en una variable si queremos hacer algo más.

```
String query = String.format(
            "INSERT INTO %s (%s,%s,%s,%s) VALUES (?,?,?,?)",
            DBScheme.TAB_COCHE, DBScheme.COL_COCHE_MATRICULA,
DBScheme.COL_COCHE_MARCA,
            DBScheme.COL_COCHE_MODELO, DBScheme.COL_COCHE_COLOR
    );
    preparedStatement = connection.prepareStatement(query);
    preparedStatement.setString(1, coche.getMatricula());
    preparedStatement.setString(2, coche.getMarca());
    preparedStatement.setString(3, coche.getModelo());
    preparedStatement.setString(4, coche.getColor());
    return preparedStatement.executeUpdate();
```

Operaciones CRUD

Cada una de las operaciones crud devuelve un tipo de dato distinto. Esto es importante, ya que dependiendo de lo que devuelva, debemos hacer una cosa u otra.

- preparedStatement.executeQuery() -> Utilizado para el SELECT. Devuelve un ResultSet que luego podremos iterar como si fuese un Array. Si no hay resultados, devolverá un objeto vacío.
- preparedStatement.executeUpdate() -> Utilizado para INSERT, DELETE y UPDATE. Devuelve un entero con el número de filas que se han modificado. Si no se ha modificado ninguna, devolverá un 0.
- preparedStatement.execute() -> NO USAR. Devuelve un booleano, pero no aporta mucha información.

findOne()

```
public Coche findById(int id) throws SQLException {
        connection = new DBConnection().getConnection();
        String query = String.format(
                "SELECT * FROM %s WHERE %s = ?",
                DBScheme.TAB_COCHE, DBScheme.COL_ID
        );
        preparedStatement = connection.prepareStatement(query);
        preparedStatement.setInt(1, id);
        resultSet = preparedStatement.executeQuery();

        ArrayList<Coche> arrayResult = getResultados(resultSet);
        if (!arrayResult.isEmpty()){
            arrayResult.get(0).showDetails();
            return arrayResult.get(0);
        } else {
            System.out.println("No hay coincidencias de búsqueda");
            return null;
        }
    }
```

Iterar sobre el resultSet

En el caso anterior, hemos hecho un executQuery, lo que nos devolverá un objeto de la clase ResultSet. Esto nos da la opción de iterarlo y guardarlo en un array desde el que explorar la consulta.

La forma de iterar sobre este resultSet es a través de un while y llamando al método .next(). Mientras haya un siguiente, seguirá iterando y podremos hacer lo que queramos con el contenido. Para acceder a los valores de cada iteración, llamaremos al nombre de la columna (utilizando nuestro DBScheme) y el tipo de dato que esperamos.

```
 private ArrayList<Coche> getResultados(ResultSet datosResultantes) throws
SQLException {
        ArrayList<Coche> listaResultado = new ArrayList<>();
        while (datosResultantes.next()){
            int id = resultSet.getInt(DBScheme.COL_ID);
            String matricula = resultSet.getString(DBScheme.COL_COCHE_MATRICULA);
            String marca = resultSet.getString(DBScheme.COL_COCHE_MARCA);
            String modelo = resultSet.getString(DBScheme.COL_COCHE_MODELO);
            String color = resultSet.getString(DBScheme.COL_COCHE_COLOR);
            listaResultado.add(mapearCoche(id, matricula,marca,modelo,color));
        }
        return listaResultado;
    }
    private Coche mapearCoche(int id, String matricula, String marca, String
modelo, String color){
        return new Coche(id,matricula,marca,modelo,color);
    }
```

Create (INSERT), Update (UPDATE) y Delete (DELETE)

Los tres funcionan de forma similar. Como nota, aquí no hay un resultSet como tal, sino que podremos retornar el número de filas modificadas.

```
    public int addNew(Coche coche) throws SQLException {
        if (findByMatricula(coche.getMatricula()) == null){
            connection = new DBConnection().getConnection();
            String query = String.format(
                    "INSERT INTO %s (%s,%s,%s,%s) VALUES (?,?,?,?)",
                    DBScheme.TAB_COCHE, DBScheme.COL_COCHE_MATRICULA,
DBScheme.COL_COCHE_MARCA,
                    DBScheme.COL_COCHE_MODELO, DBScheme.COL_COCHE_COLOR
            );
            preparedStatement = connection.prepareStatement(query);
            preparedStatement.setString(1, coche.getMatricula());
            preparedStatement.setString(2, coche.getMarca());
            preparedStatement.setString(3, coche.getModelo());
            preparedStatement.setString(4, coche.getColor());
            return preparedStatement.executeUpdate();
        }
        System.out.println("Ya existe un coche con la ID indicada");
        return 0;
        }
```