1. BBDD Relacional

Una *base de datos relacional* (BDR) es un sistema organizado de datos que representa la información conforme al modelo relacional y que garantiza la integridad referencial. Está compuesta de un conjunto de tablas en las que se almacena la información de cada una de las entidades y las relaciones existentes entre ellas.

Existe una clave primaria que se corresponde con uno o varios atributos (consideramos que cada atributo se corresponde con una columna), que permite identificar de forma unívoca cada tupla.

Este tipo de bases de datos son ampliamente utilizadas y existe un gran número de alternativas tanto libres como propietarias.

Todo lo que se comentará en esta guía es compatible con *MySQL* (propietaria) y *MariaDB* que es una alternativa libre.

Desde Java utilizaremos *JDBC* (Java Database Connectivity). Esta es la 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.

2. Establecimiento de la conexión

Para poder conectarse a una base de datos y lanzar consultas, una aplicación necesita tener un *conector* adecuado. Un conector suele ser un fichero *.jar* que contiene una implementación de todas las interfaces del API JDBC.

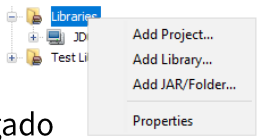
> Un conector o driver es un conjunto de clases encargadas de implementar las interfaces del API y acceder a la base de datos

Cada SGBD tiene un driver específico que podemos descargar desde la página de cada fabricante.

> El conector para MariaDB se descarga desde https://downloads.mariadb.org/connector-java/

Una vez descargado el driver, deberá añadirse al proyecto como una librería. Desde NeatBeans:

1. Pulsar con botón derecho sobre *Libraries* (librerías del proyecto)
2. Se mostrará un menú con varias opciones. Seleccionar *Add JAR/Folder...*
3. Buscar en nuestras carpetas el archivo .jar que hemos descargado previamente.



Entre nuestro programa java y el Sistema Gestor de Bases de Datos (SGBD) se intercala el conector JDBC. Este conector es el que implementa la funcionalidad de las clases de acceso a datos y proporciona la comunicación entre el API JDBC y el SGBD.

A continuación, se registrará desde el código del programa y se hará la conexión con la base de datos

```
try {
    Class.forName("org.mariadb.jdbc.Driver");
    String jdbcUrl = “jdbc:mariadb://localhost:3306/profesores”;
    Connection con = DriverManager.getConnenction(jdbcUrl,"username","password");
    …
} catch(SQLException e){
    System.out.println("SQL Exception: "+ e.toString());
} catch (ClassNotFoundException cE) {
    System.out.println("Excepción: "+ cE.toString());
}
```

Como se ve en el ejemplo, en Java podemos utilizar el método *getConnection()* de la clase *DriverManager* para establecer una conexión con una base de datos. Este método recibe como parámetro la *URL de JDBC* que identifica a la base de datos con la que queremos realizar la conexión.

La ejecución de este método devuelve un objeto *Connection* que representa la conexión con la base de datos.

Cuando se presenta con una URL específica, DriverManager itera sobre la colección de drivers registrados hasta que uno de ellos reconoce la URL especificada. Si no se encuentra ningún driver adecuado, se lanza una *SQLException*.

A partir de esto, podemos utilizar los siguientes tipos de sentencias:

Statement: para sentencias sencillas en SQL.
PreparedStatement: para consultas preparadas, como por ejemplo las que tienen parámetros.

El API JDBC distingue dos tipos de consultas:

Consultas de lectura: SELECT. Para las sentencias de consulta que obtienen datos de la base de datos, se emplea el método *executeQuery(String sql)*. El método de ejecución del comando SQL devuelve un objeto de tipo *ResultSet* que sirve para contener el resultado del comando SELECT, y que nos permitirá su procesamiento.

Consultas de Actualización: INSERT, UPDATE, DELETE, *sentencias DDL*(utilizadas para la creación de una base de datos, tablas,⋯). Para estas sentencias se utiliza el método *executeUpdate(String sql).*

3. Ejecución de consultas

Las consultas a la base de datos se realizan con sentencias SQL que van "*embebidas*" en otras sentencias especiales que son propias de Java. Por tanto, podemos decir que las consultas SQL las escribimos como parámetros de algunos métodos Java que reciben el String con el texto de la consulta SQL.

Las consultas devuelven un *ResultSet*, que es una clase java parecida a una lista en la que se aloja el resultado de la consulta. Cada elemento de la lista es uno de los registros de la base de datos que cumple con los requisitos de la consulta.

El *ResultSet* no contiene todos los datos, sino que los va obteniendo de la base de datos según se van pidiendo. La razón de esto es evitar que una consulta que devuelva una cantidad muy elevada de registros, tarde mucho tiempo en obtenerse y sature la memoria del programa.

Para realizar cualquier tipo de operación se utilizará el objeto *Statement*. Este, a través del método *executeQuery*, ejecutará la consulta SQL que recibe como parámetro y recuperará los datos en un objeto *ResulSet*.

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM profes");
```

Algunos de los métodos que ofrece *ResultSet* para el manejo de los resultados son:

- *next().-* Para pasar al siguiente registro y seguir leyendo la información. Devuelve false cuando no hay más registros.
- *previuos().- Comportamiento inverso al de next().*
- *first().-*posicionarse en el primer registro.
- *last().-*posicionarse en el último registro.
- *getXXX(“*campo|columna*”).-* obtiene la información de los campos de la fila actual, puede emplearse el nombre o el número de la columna (comienza en 1) . Debe indicarse el tipo de dato que se va a recuperar.

getInt - INTEGER	getBigDecimal - DECIMAL	getDate - DATE
getLong - BIG INT	getBoolean - BIT	getTime - TIME
getFloat - REAL	getString - VARCHAR	getTimestamp - TIME STAMP
getDouble - FLOAT	getString - CHAR	getObject - OTROS

Una vez que tenemos el resultado de la consulta en el *ResultSet*, deberemos extraer los datos para su procesado.

Lo habitual es recorrer uno a uno los registros obtenidos hasta el final, para ello crearemos un bucle que finalizará cuando no queden más elementos: while (rs.next())

Supongamos que lo que se pide es un listado de todos los datos :

```
while (rs.next()){
   System.out.println(rs.getString(1) + " " + rs.getString(2));
}
```

```
while (rs.next()){
   System.out.println(rs.getString(“dato1”) + " " + rs.getString(”dato2”));
}
```

Ambas soluciones son válidas, haciendo el *get* desde el número de columna o desde su nombre. No debemos olvidar usar el método get adecuado para cada tipo de dato.

A continuación, se muestra un ejemplo de lectura de la tabla de *profesores* que forma parte de la base de datos *profesorado*. Para este ejemplo utilizaremos el *SGBD* MaríaDB.

```
String bd = "profesorado";
String user = "root";
String password = "";
String url = "jdbc:mariadb://localhost:3306/" + bd;
String drive = "org.mariadb.jdbc.Driver";
Connection con = null;

try {
    1- Cargar el Driver del SGBD
    Class.forName(drive);
    2- Crear la conexión, el formato de la URL depende de SGBD
    con = (Connection) DriverManager.getConnection(url, user, password);
} catch (ClassNotFoundException ex) {
    System.out.println("Falta el driver" + ex.getMessage());
} catch (SQLException ex) {
    System.out.println(ex.getMessage());
}
try {
    3- Necesitamos un objeto (Profesor) y un ArrayList para guardar los objetos
    Profesor prf;
    ArrayList<Profesor> profes = new ArrayList<>();

    4- Creamos la sentencia
    Statement stmt = con.createStatement();
    5- Obtenemos el resultado de la consulta
    ResultSet rs = stmt.executeQuery("SELECT * FROM profesores");

    6- Recorremos el ResultSet obtenido
    while (rs.next()) {
        prf = new Profesor(rs.getString("dni"),
                           rs.getString("nombre"),
                           rs.getInt("edad"),
                           rs.getString("departamento"));
    7- Después de crear el objeto lo almacenamos en memora, en este caso en un ArrayList
        profes.add(prf);
    }

    8- Gestionamos o procesamos los objetos, en este caso los mostramos por pantalla
    for (Profesor pr : profes) {
        System.out.println(pr);
    }
} catch (SQLException ex) {
    System.out.println(ex.getMessage());
}
```

4. Actualización de información

Como para las consultas vistas anteriormente, debemos establecer una conexión a la base de datos para utilizar un objeto *Statement* con el que enviaremos una consulta mediante el método *executeUpdate*. Este método devuelve el número de filas afectadas.

Ejemplo de como añadir un nuevo registro a nuestra base de datos:

```
Statement stmt = con.createStatement();

int cantidad = stmt.executeUpdate("INSERT INTO profes  (dni, nombre, edad,
departamento ) VALUES ( '1234678Z',  'Pedro', 35,  'Informática')");
```

o también

```
int cantidad = stmt.executeUpdate("INSERT INTO profes VALUES ('1234678Z',
'Pedro', 35,  'Informática')");
```

también puede utilizarse esta sintaxis

```
int cantidad = stmt.executeUpdate("INSERT INTO profesores SET "
            + "dni = '123dasg', "
            + "nombre = 'Ana', "
            + "edad = 32, "
            + "departamento = 'Historia'");
```

El funcionamiento de las modificaciones y borrados es idéntico al visto para creación, donde solo variará la consulta que se envía:

```
Statement stmt = con.createStatement();
int cantidad = stmt.executeUpdate("UPDATE profesores SET "
            + "nombre = 'María' "
            + "WHERE dni = '123dasg'");
```

Recuerda que la cláusula *WHERE* es necesaria para modificaciones y borrados, sino ejecutará la instrucción sobre TODOS los registros de la tabla.

Finalmente, para borrar tendremos el siguiente código:

```
Statement stmt = con.createStatement();
int cantidad = stmt.executeUpdate("DELETE FROM profesores WHERE dni =
'123dasg'");
```

5. Sentencias preparadas

Las sentencias preparadas de JDBC permiten la *precompilación* del código SQL antes de ser ejecutado, permitiendo consultas o actualizaciones más eficientes.

Las ventajas de usar sentencias preparadas son básicamente dos:

- En el momento de compilar la sentencia SQL, se analiza cuál es la estrategia adecuada según las tablas, las columnas, los índices y las condiciones de búsqueda implicados. Este proceso, obviamente, consume tiempo de procesador, pero al realizar la compilación una sola vez, se logra mejorar el rendimiento en siguientes consultas iguales con valores diferentes.
- Permiten la parametrización: la sentencia SQL se escribe una vez, indicando las posiciones de los datos que van a cambiar y, cada vez que se utilice, le proporcionaremos los argumentos necesarios que serán sustituidos en los lugares correspondientes. Los parámetros se especifican con el carácter '?'.

Lo primero que haremos será ***preparar*** la instrucción SQL. Utilizaremos tantas interrogaciones como elementos variables tendrá la sentencia.

Por ejemplo, nuestra tabla de *profesores* consta de 4 columnas o campos. El *INSERT* siempre es igual, varían los datos que introduciremos, en este caso 4.

```
String sql = "INSERT INTO profesores VALUES ( ?, ?, ?, ? )";
PreparedStatement sentencia = conexion.prepareStatement(sql);
```

Ahora, para cada registro debemos agregar los valores

```
sentencia.setString(1, txtDni);
sentencia.setString(2, txtNombre);
sentencia.setInt(3, txtEdad);
sentencia.setString(4, txtDepartamento);
sentencia.executeUpdate();
```

6. Transacciones

Las *transacciones* son conjuntos de operaciones de actualización en una base de datos que, al ser consideradas como una operación múltiple de mayor alcance que cada uno de sus componentes individuales, deben realizarse de forma completa o nula.

Cualquiera que estudie la teoría de cómo funcionan las SXBDR se encontrará con el acrónimo *ACID*, útil para recordar las cuatro características que debe proporcionar una transacción.

- *Atomicidad*; cada cambio que es parte del conjunto es exitoso o la base de datos debe permanecer en su estado anterior. No se permiten cambios parciales.
- *Coherencia*; la base de datos no puede dejarse en un estado inconsistente, con información que pierde su significado.
- *Aislamiento*; las diferentes transacciones no pueden interferir entre sí.
- *Durabilidad*; las transacciones debidamente completadas deben garantizar la persistencia de los datos.

Por defecto, una conexión funciona en modo *autocommit*, es decir, cada vez que se ejecuta una sentencia SQL se abre y se cierra automáticamente una transacción que afecta únicamente a dicha sentencia.

Los métodos *setAutoCommit*() y *getAutoCommit*() permiten modificar y conocer como está autocommit activo o no.

Si no se está trabajando en modo autocommit será necesario cerrar **explícitamente** las transacciones mediante *commit*() si tienen éxito o *rollback*() si fallan.

También es posible especificar el nivel de aislamiento de una transacción mediante *setTransactionIsolation*(). Los niveles de aislamiento se representan mediante las constantes que se muestran en la lista siguiente:

```
mysqlCon.setAutoCommit(false);
try{
    …
    stmt.executeUpdate ();
    mysqlCon.commit();
} catch (SQLException e ){
```

```
        System.err.print("Ha habido problemas, se deshará la transacción");
        mysqlCon.rollback();
    }
```

- *TRANSACTION_NONE*, no se permiten realizar transacciones.
- *TRANSACTION_READ_UNCOMMITED*, desde esta transacción se pueden llegar a ver registros que han sido modificados por otra transacción, pero no guardados, por lo que es posible llegar a trabajar con valores que nunca lleguen a guardarse realmente.
- *TRANSACTION_READ_COMMITED*, se ven sólo las modificaciones ya guardadas por otras transacciones.
- *TRANSACTION_REPETEABLE_READ*, si se leyó un registro, y otra transacción lo modifica guardándolo y se vuelve a leer, se seguirá viendo la información que había cuando se leyó por primera vez. Es decir, evita la situación en que una transacción lee una fila, una segunda transacción altera esa fila y la primera transacción vuelve a leer la fila, obteniendo valores diferentes a la primera lectura. Esto proporciona un mayor nivel de consistencia.
- *TRANSACTION_SERIALIZABLE*, se verán todos los registros tal y como estaban antes de comenzar la transacción, independientemente de las modificaciones que otras transacciones hagan ni de que se haya leído antes o no. Si se añadió algún nuevo registro, tampoco se verá.