

UT-12: ACCESO A BASES DE DATOS RELACIONALES

¿Qué vamos a ver?

- Bases de datos relacionales. Características.
 - Establecimiento de conexiones.
 - Recuperación de información.
 - Manipulación de la información.
 - Ejecución de consultas sobre la base de datos.

1. BASES DE DATOS RELACIONALES. CARACTERÍSTICAS.

Una base de datos relacional es un tipo de base de datos que almacena y proporciona acceso a puntos de datos relacionados entre sí. Las bases de datos relacionales se basan en el modelo relacional, una forma intuitiva y directa de representar datos en tablas. En una base de datos relacional, cada fila en una tabla es un registro con uno o varios campos que utilizamos para identificarlo del resto, a lo que llamamos clave. Las columnas de la tabla contienen los atributos de los datos y cada registro suele tener un valor para cada atributo, lo que simplifica la creación de relaciones entre los puntos de datos.

El modelo relacional significa que las estructuras de datos lógicas (las tablas de datos, las vistas y los índices) están separadas de las estructuras de almacenamiento físicas. Gracias a esta separación, los administradores de bases de datos pueden gestionar el almacenamiento físico de datos sin que eso influya en el acceso a esos datos como estructura lógica.

La distinción entre lógico y físico se aplica también a las operaciones de base de datos, que son acciones claramente definidas que permiten a las aplicaciones manipular los datos y las estructuras de la base de datos. Con las operaciones lógicas, las aplicaciones pueden especificar el contenido que necesitan, mientras que las operaciones físicas determinan cómo se debe acceder a esos datos y llevan a cabo la tarea.

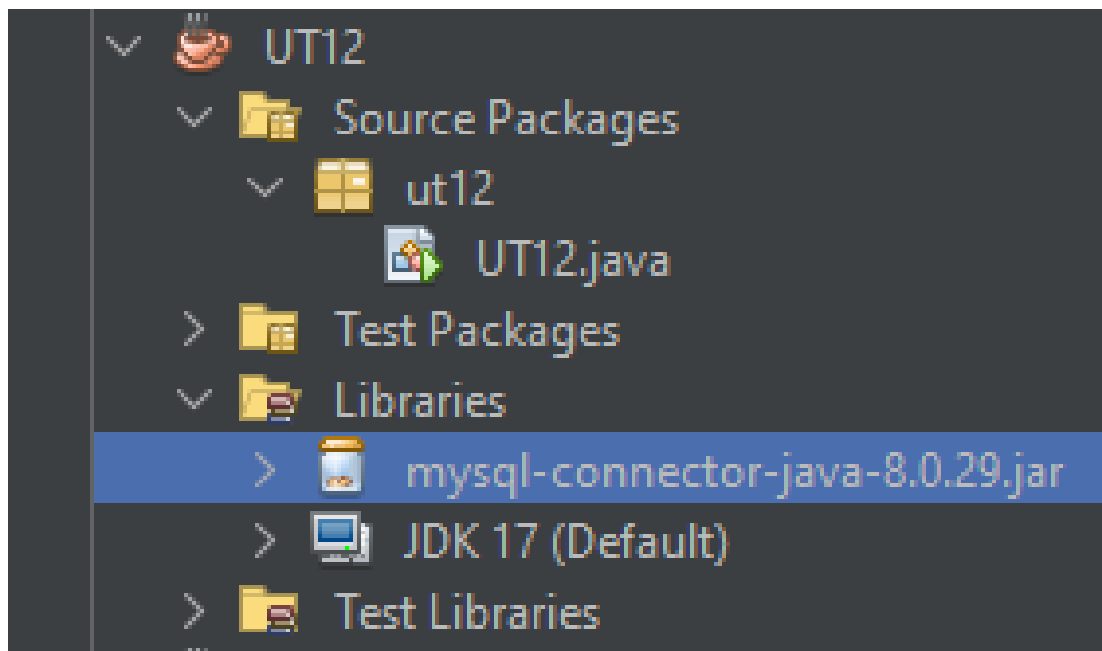
Para garantizar la precisión y accesibilidad continua de los datos, las bases de datos relacionales siguen ciertas reglas de integridad.

Gracias a la librería `java.sql` nos será posible conectarnos a bases de datos que sigan dicho estándar.

En nuestro caso nos vamos a conectar a bases de datos MySQL. Para ello necesitaremos importar en nuestro proyecto una librería específica para realizar las operaciones de conexión, recuperación de información, manipulación de información o ejecución de consultas.

1.1. Establecimiento de conexiones

Como hemos comentado anteriormente, para poder conectarnos a la base de datos MySQL lo primero que debemos hacer es incorporar la librería específica para ello. En nuestro caso se tratará de `mysql-connector-java-8.0.29`.



Una vez tengamos esta librería en nuestro proyecto realizaremos la conexión a la base de datos por medio de código.

Lo primero que vamos a necesitar es una cadena de conexión, que para esta librería necesita que tenga la siguiente estructura:

```
String cadenaConexion = "jdbc:mysql://host:port/dbName";
```

Donde:

- host: el servidor al que nos conectaremos (para nosotros será localhost)

- port: el puerto específico del servidor al que nos conectaremos (para mysql habitualmente es 3306)
- dbName: el nombre de la base de datos sobre la que queremos operar.

Un ejemplo de cadena de conexión sería el siguiente:

```
String cadenaConexion =  
"jdbc:mysql://localhost:3306/pruebasprogramacion";
```

Después vamos a necesitar cargar el Driver de mysql y el conector jdbc. Esto se lleva a cabo mediante la siguiente instrucción.

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

Una vez hayamos hecho estos preparativos ya estamos listos para realizar la conexión con la base de datos. Esto requerirá de que importe las clases `java.sql.Connection` y `java.sql.DriverManager`.

Con el método estático `getConnection` vamos a conseguir una conexión a la base de datos y la almacenaremos en una variable de tipo `Connection`.

```
Connection conexion = DriverManager.getConnection(cadenaConexion,  
nombreUsuario, password);
```

A este método le tenemos que pasar como parámetros la cadena de conexión que creamos previamente, el nombre del usuario con el que nos vamos a conectar y la contraseña del mismo.

Si todo va bien, esto conectará con la base de datos y ya podremos operar con ella.

Puesto que es posible que salte alguna excepción, debemos recubrir esto con un `try...catch`, para poder manejarlas de una forma controlada.

Al igual que ocurría con los ficheros, es necesario que una vez que terminemos de utilizar nuestra conexión a la base de datos la cerremos. Esto lo haremos en el `finally` de `try...catch`, como hacíamos con los ficheros.

Veamos un ejemplo de conexión y desconexión a una base datos MySQL.

```
import java.sql.DriverManager;
import java.sql.Connection;

public class UT12 {

    public static void main(String[] args) {

        String cadenaConexion =
"jdbc:mysql://localhost:3306/pruebasprogramacion";
        Connection conexion = null;

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            conexion = DriverManager.getConnection(cadenaConexion,
"root", "");
            if (conexion != null)
                System.out.println("Me he conectado a la base de datos");
        } catch (Exception e) {
            System.out.print("Ha ocurrido un error al intentar
conectarme: ")
            System.out.println(e.toString());
        } finally{
            try {
                if(conexion != null){
                    conexion.close();
                }
            } catch (Exception e2) {
                System.out.println(e2.toString());
            }
        }

    }

}
```

1.2. Recuperación de la información

Crear una conexión a una base de datos no tiene mucha utilidad por si solo, es decir, se trata solo del paso previo para realizar las operaciones importantes, recuperación y modificación de la información.

En este apartado nos vamos a centrar en la recuperación de la información.

1.2.1. Recuperación de la información de forma no parametrizada

La primera forma que vamos a ver de ejecutar consultas solicitando información a la base de datos va a ser mediante la clase `java.sql.Statement`.

Esta clase nos va a permitir lanzar una consulta escrita en una cadena de caracteres y luego vamos a recoger los datos que nos devuelve la base de datos en un objeto de otra nueva clase, `java.sql.ResultSet`, que representa todas las filas que recogemos como resultado de la consulta.

Primero vamos a crear el objeto `Statement` a través del método `createStatement()` del objeto `Connection` que ya tenemos.

Después ejecutaremos la consulta con el método `ResultSet executeQuery(String consulta)` del objeto `Statement` y lo almacenaremos en un objeto `ResultSet`.

Una vez que tengamos el objeto `ResultSet`, lo recorreremos fila por fila, pudiendo extraer los datos de cada una de ellas, hasta que no queden filas.

Finalmente, cerraremos la conexión.

Veamos un ejemplo:

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

public class UT12 {

    public static void main(String[] args) {

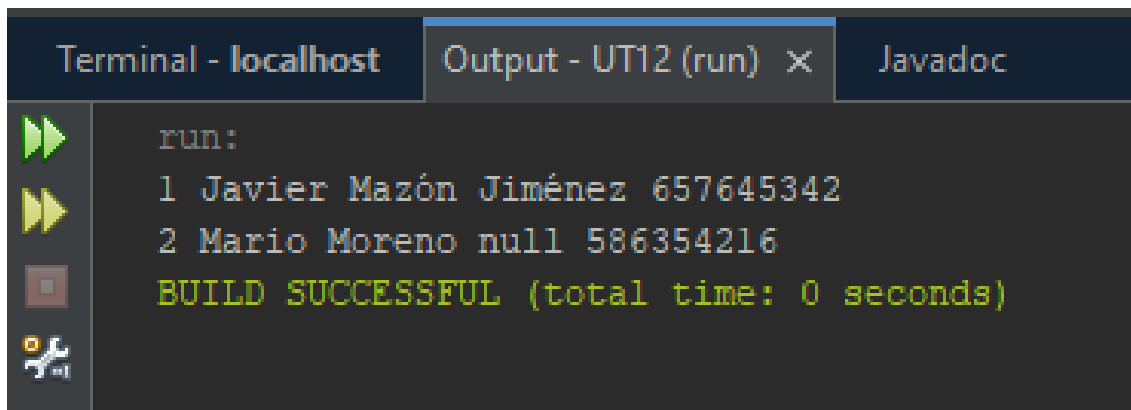
        String cadenaConexion =
"jdbc:mysql://localhost:3306/pruebasprogramacion";
        String consulta = "SELECT * FROM Tabla1";
        Connection conexion = null;

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
```

```
        conexion = DriverManager.getConnection(cadenaConexion,
"root", "");
        Statement state = conexion.createStatement();
        ResultSet result = state.executeQuery(consulta);

        while(result.next()){
            System.out.print(result.getInt("id") + " ");
            System.out.print(result.getString("nombre") + " ");
            System.out.print(result.getString("apellido1") + " ");
            System.out.print(result.getString("apellido2") + " ");
            System.out.println(result.getInt("telefono") + " ");
        }
    } catch (Exception e) {
        System.out.print("Ha ocurrido un error al intentar
conectarme: ")
        System.out.println(e.toString());
    } finally{
        try {
            if(conexion != null){
                conexion.close();
            }
        } catch (Exception e2) {
            System.out.println(e2.toString());
        }
    }
}
}
```

Esto nos daría como resultado por consola lo siguiente:



Utilizaremos un bucle para ir recorriendo todas las filas, y el método `next()` del objeto `ResultSet` para recoger la siguiente fila. Este método nos devolverá un `false` en el caso de que ya no queden más filas por leer.

Dentro del bucle, iremos leyendo los distintos datos de la fila con métodos como:

- Integer **getInt** (String nombreColumna)
- String **getString** (String nombreColumna)
- Double **getDouble** (String nombreColumna)
- Float **getFloat** (String nombreColumna)
- Boolean **getBoolean** (String nombreColumna)
- Date **getDate**(String nombreColumna)
- Time **getTime**(String nombreColumna)
- Timestamp **getTimestamp**(String nombreColumna)

Fíjate que los métodos no nos devuelven tipos primitivos de datos, sino sus versiones wrapper, ya que nos puede llegar un valor nulo y al recoger el dato nos podría dar un error. Existe un método para comprobar si el último dato que hemos recogido es un nulo, por lo que no necesitamos comprobarlo antes. Este método es `isNull()` y lo ejecutaremos después de cualquier de los métodos `get` que acabamos de ver.

Los tipos `Date`, `Time` y `DateTime` de `java.sql` no son compatibles con `LocalDate`, `LocalTime` y `LocalDateTime`, por lo que vamos a necesitar realizar casteos a la hora guardarlos en objetos de estos tipos. De hecho, ni siquiera existe un tipo `DateTime` de `java.sql`, y por lo que utilizaremos el método `getTimestamp` para recoger este tipo de datos.

```
LocalTime hora = rs.getTime("hora").toLocalTime();
LocalDate fecha = rs.getDate("fecha").toLocalDate();
LocalDateTime fechaHora = rs.getTimestamp("fechaHora").toLocalDateTime();
```


Por supuesto a un objeto Statement le podemos pasar una consulta que contenga filtros, subconsultas, etc, por ejemplo:

```
String consulta = "SELECT * FROM Tabla1 WHERE id = 1";
```

1.2.1. Recuperación de la información de forma parametrizada

La opción no parametrizada de recuperación de la información, aunque no sea la más recomendable, la podemos utilizar cuando es el propio programador el que especifica la consulta de forma completa, y no utiliza datos que le proporcione el usuario u otro sistema para construir dicha consulta.

El problema viene cuando lo que necesitamos es utilizar datos proporcionados por el usuario u otros sistemas para construir la consulta, algo similar a esto:

```
String consulta = "SELECT * FROM Tabla1 WHERE id = " + id;
```

Siendo la variable id proporcionada por el usuario.

Construir las consultas de esta forma implica un gran riesgo debido a que podemos ser víctimas de un ataque muy común denominado inyección de código. Este ataque se basa en introducir en campos de texto parte de sentencias SQL para conseguir como resultado la extracción de datos de la base de datos, o conseguir cualquier otro fin malintencionado.

Debido a esto, es mucho más recomendable la utilización de la forma parametrizada, la cual nos protege de este tipo de situaciones.

La forma parametrizada consiste en crear un objeto `java.sql.PreparedStatement`, al que le pasaremos una consulta como String, pero cambiaremos los valores que utilizemos para los filtros por el signo de interrogación (?). Después iremos asignado a cada una de esas interrogaciones un tipo de dato y un valor, que el `PreparedStatement` se encargará de combinar de forma segura.

Una vez hecho esto, ya solo tendremos que ejecutar la consulta.

Veamos un ejemplo de cómo se hace:

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.PreparedStatement;

public class UT12 {

    public static void main(String[] args) {

        String cadenaConexion =
"jdbc:mysql://localhost:3306/pruebasprogramacion";
        String consulta = "SELECT id, nombre, apellido1, apellido2,
telefono FROM tabla1 WHERE id <> ?";
        Connection conexion = null;

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            conexion = DriverManager.getConnection(cadenaConexion,
"root", "");
            PreparedStatement ps = conexion.prepareStatement(consulta);
            ps.setInt(1, 2); // primer campo posición, segundo campo
valor

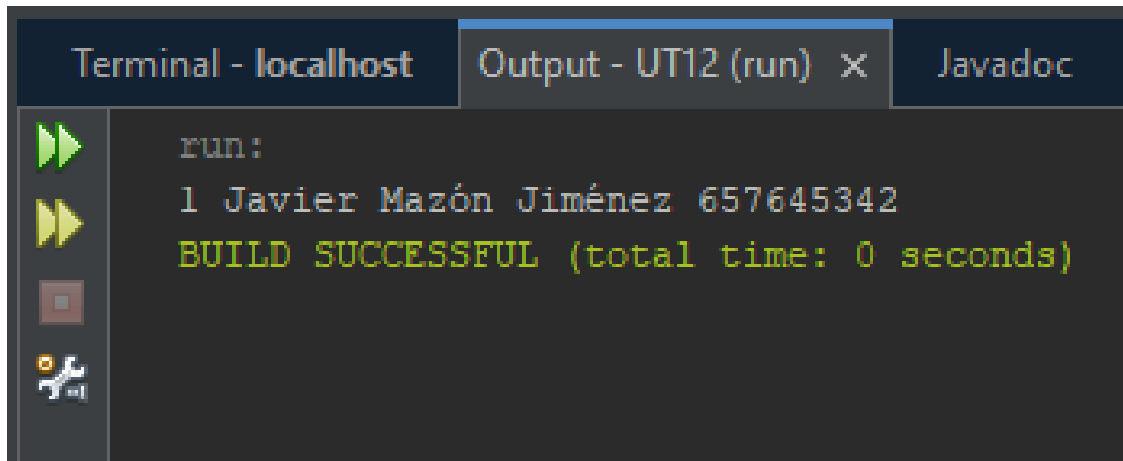
            ResultSet result = ps.executeQuery();

            while(result.next()){
                System.out.print(result.getInt("id") + " ");
                System.out.print(result.getString("nombre") + " ");
                System.out.print(result.getString("apellido1") + " ");
                System.out.print(result.getString("apellido2") + " ");
                System.out.println(result.getInt("telefono") + "
");
            }

        } catch (Exception e) {
            System.out.println(e.toString());
        } finally{
            try {
                if(conexion != null){
                    conexion.close();
                }
            } catch (Exception e2) {
                System.out.println(e2.toString());
            }
        }

    }

}
```

A screenshot of an IDE terminal window. The window has three tabs: 'Terminal - localhost', 'Output - UT12 (run) x', and 'Javadoc'. The 'Output - UT12 (run) x' tab is active. The terminal shows the following output: 'run:' followed by '1 Javier Mazón Jiménez 657645342' and 'BUILD SUCCESSFUL (total time: 0 seconds)' in green text. On the left side of the terminal, there are icons for running (a green play button), debugging (a yellow play button), and other IDE functions (a red square and a wrench icon).

Como podemos ver, una vez que creamos el objeto `PreparedStatement`, mediante el método `prepareStatement(String consulta)`, a continuación vamos a dar valor a las interrogaciones que aparecen en la consulta. Para ello utilizaremos métodos como:

- **`setString(pos, valor)`**
- **`setInt(pos, valor)`**
- **`setBoolean(pos, valor)`**
- **`setFloat(pos, valor)`**
- **`setDouble(pos, valor)`**
- **`setDate(pos, valor)`**
- **`setTime(pos, valor)`**

Al igual que ocurre con los tipos de datos de tiempo y fecha al recogerlos, al introducirlos también vamos a necesitar castearlos de la siguiente manera:

```
LocalTime hora = LocalTime.now();
ps.setTime(1, Time.valueOf(hora));

LocalDate fecha = LocalDate.now();
ps.setDate(2, Date.valueOf(fecha));

LocalDateTime fechaHora = LocalDateTime.now();
ps.setString(3, fechaHora.toString());
```

En el caso de los objetos de tipo `LocalDateTime` los vamos a pasar directamente como un `String`.

Después de pasar los parámetros, ejecutaremos el método `executeQuery()` del objeto `PreparedStatement` y recogeremos los datos en un `ResultSet` de la misma forma que lo hacíamos con los `Statement`.

Ya va siendo hora de que practiquemos con nuestras primeras consultas sobre bases de datos:

