CRUD BÁSICO

Por Mikel Jorge y Eneko Martinez

Vamos a crear un programa que implemente la estructura y clases básicas CRUD y posteriormente lo probaremos. En ningún momento nos preocuparemos de validar los datos ya que de eso se encargará posteriormente la capa de servicios.

Completa las partes indicadas en color rojo. Deberás entregar al finalizar el proyecto completo en un **.zip** con el nombre requerido en la entrega.

Antes de empezar…

Vamos a crear la base de datos a la que accederemos en el programa. Para ello, usaremos ***WORKBENCH*** y el script de creación de la base de datos proporcionado como recurso en la actividad: *Script_creacion_bdbasicas.sql*.

Abrimos el script en Workbench y tras ejecutarlo y actualizar veremos creada nuestra base de datos `bdbasicas`. Esta será la base de datos a la que nos conectaremos.

Creación del proyecto

Primero creamos el proyecto en el IDE. Será un proyecto **MAVEN** similar al de la actividad guiada *UT2_A3*.

Añadiremos las dependencias al archivo *pom.xml*, en este caso usaremos las del conector de MYSQL y las del logger.

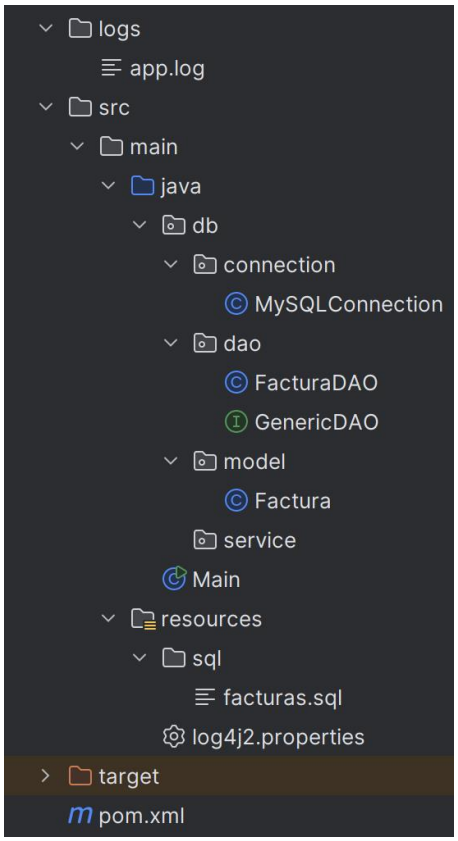
```
<dependencies>
    <!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>8.3.0</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-api -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>2.24.1</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.24.1</version>
    </dependency>
</dependencies>
```

Recuerda actualizar las dependencias del archivo mediante el botón que aparece en la esquina superior derecha del *pom.xml*.

Nuestro proyecto deberá tener este aspecto al acabar.



¿Qué queremos que haga nuestro programa?

Nuestro programa va a crear una tabla en nuestra base de datos y a poblarla de datos mediante la ejecución de un script (si así lo deseamos). Seguidamente haremos uso de los métodos de la clase DAO para hacer operaciones CRUD sobre los datos de nuestra tabla. Sacaremos por `System.out` los resultados de nuestras operaciones. Tendremos que llevar buena cuenta de todas las operaciones y/o problemas mediante el logger.

Creación de la clase de instancia única para conectar a la base de datos

Creamos esta clase llamada `MYSQLConnection` de manera similar a como lo hicimos en *UT2_A3 Actividad Guiada*, teniendo en cuenta que el nombre de nuestra base de datos ha cambiado.

También debemos añadir a nuestra conexión la configuración extra que nos permitirá ejecutar consultas múltiples, tal como aparece en la presentación que trata el tema.

```
// Datos de conexión
private final String SERVER = "localhost";  1 usage
private final String PORT = "3309";  1 usage
private static final String DB = "bdbasicas";  4 usages
private final String DBMS = "mysql";  1 usage
private final String URL = "jdbc:" + DBMS + "://"+ SERVER +":" + PORT + "/" + DB + "?allowMultiQueries=true";
private final String USER = "root";  1 usage
private final String PASSWORD = "root";  1 usage
```

Creación del modelo

Crearemos la clase que modela la tabla `Facturas` de nuestra base de datos. Estudiaremos la estructura de la tabla, sus columnas y el tipo de datos de cada una de ellas.

Column	Type	Default Value	Nullable
codigo	int		NO
cuenta	int		NO
destinatario	varchar(90)		NO
fecha_hora	datetime		NO
importe	decimal(10,2)		NO

Con esto sabremos qué atributos tendrá nuestra clase `Facturas` que servirá para modelar la tabla. Para la columna `fecha_hora` usaremos de tipo la clase `LocalDateTime`

```
package db.model;

import ...

public class Factura {  17 usages
    private int codigo;  4 usages
```

1. Completa la clase con los atributos necesarios, el constructor, *getters* y *setters*, y el método *@override toString()* que usaremos para formatear la impresión de los datos guardados.

Nota: para formatear el `LocalDateTime` *con la fecha y hora de manera que quede similar a lo almacenado en la base de datos, podemos usar algo similar a esto. También podríamos cambiar el orden de la fecha para que aparezca como lo usamos normalmente.*

```
fechaHora.format(DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm:ss"));
```

Creación de las clase DAO

Seguidamente tendremos que crear la clase DAO para `Facturas` que realice las operaciones CRUD sobre la tabla. Para ello crearemos `FacturasDAO`. Sin embargo, antes vamos a crear una interfaz genérica `GenericDAO` con los métodos de las operaciones CRUD que implementarán después el resto de nuestras clases DAO (en el caso de esta actividad sólo será una clase, `FacturasDAO`). La clase genérica tendrá el mismo aspecto que hemos visto en la presentación

```
package db.dao;

import ...

public interface GenericDAO<T, K> {  1 usage  1 implementation

    // Método para insertar un nuevo registro (CREATE)
    void insertar(T entity) throws SQLException;  1 usage  1 implem

    // Método para obtener todos los registros (READ)
    List<T> obtenerTodos() throws SQLException;  1 usage  1 impleme

    // Método para obtener un registro por ID (READ)
```

Al implementar la interfaz `genericDAO` en nuestra clase `FacturasDAO` sustituiremos *T* con la clase de nuestro modelo (`Factura`) y *K* con el tipo de la clave primaria de nuestra clase que

nos sirva para identificar unívocamente los registros (*código en nuestro caso*, y por lo tanto su tipo será `Integer`).

```
package db.dao;

import ...

public class FacturaDAO implements GenericDAO<Factura, Integer> {
```

Ahora deberíamos empezar a sobreescribir los métodos CRUD para la clase de nuestro modelo, o sea, `Factura`.

Método *obtenerTodos()*

Vamos a ver un ejemplo con el método `obtenerTodos()`. Este método nos devolverá una lista de objetos `Factura` con todos los registros de la tabla y que es una de las dos operaciones de tipo *Read* que escribiremos (la otra es `obtenerPorId(Integer codigo)` que nos devolverá un solo registro).

```
@Override  1 usage
public List<Factura> obtenerTodos() throws SQLException {
    List<Factura> Facturas = new ArrayList<>();
    Connection connection = MySQLConnection.getInstance().getConnection();
```

Creamos la variable de retorno y obtenemos la conexión a la base de datos.

Seguidamente creamos un `Statement` (no usaremos `PreparedStatment` aún) en nuestra conexión que será el que ejecute nuestra consulta, la cual escribiremos a continuación. Esta consulta nos devuelve una tabla con todos los registros, que almacenamos en un objeto tipo `ResultSet`.

```
    Statement stmt = connection.createStatement();
    ResultSet rsFacturas = stmt.executeQuery( sql: "SELECT * FROM Facturas");
```

Recorremos el `ResultSet` fila a fila recuperando de cada una de ellas los datos de un registro y creando un objeto de nuestra clase `Factura` que añadimos a la lista que retornaremos al final. Tendremos cuidado de usar el método adecuado para cada tipo de dato al recuperar los datos de cada columna del `ResultSet`.

```
    while (rsFacturas.next()) {
        Facturas.add(new Factura(
                rsFacturas.getInt( columnLabel: "codigo"),
                rsFacturas.getString( columnLabel: "destinatario"),
                rsFacturas.getInt( columnLabel: "cuenta"),
                rsFacturas.getDouble( columnLabel: "importe"),
                rsFacturas.getDate( columnLabel: "fecha_hora").toLocalDate().atTime(rsFactur
    }
    return Facturas;
```

`ResultSet.getDate(key).toLocalDate().atTime(ResultSet.getTime(key).toLocalTime())` devuelve un `LocalDateTime` con la fecha y hora recuperadas del `ResultSet`.

```
rsFacturas.getDate( columnLabel: "fecha_hora").toLocalDate().atTime(rsFacturas.getTime( columnLabel: "fecha_hora").toLocalTime()));
```

Con esto ya tenemos nuestro método listo. Eso sí, debemos recordar (no como yo) cerrar nuestro Statement para recuperar recursos.

2. Completa el resto de métodos CRUD: *insertar(Factura), obtenerPorId(Integer), actualizar(Factura) y eliminar(Integer).*

Código SQL: *Este es el código de las operaciones CRUD*

```
INSERT INTO Facturas (codigo, cuenta, destinatario, fecha_hora, importe)
       VALUES (int, int, 'String', 'LocalDateTime', double)

SELECT * FROM Facturas WHERE codigo = int

UPDATE Facturas SET cuenta=int,
              destinatario='String',
              fecha_hora = 'LocalDateTime',
              importe = double
              WHERE código = int

DELETE FROM Facturas WHERE código = int
```

Al preparar las consultas en el `Statement`, recuerda poner todas las comas y comillas entre comillas dobles.

Con respecto a `LocalDateTime`, para insertar debemos formatearla para que coincida con lo que MySQL espera. Para ello usaremos lo siguiente

```
Factura.getFechaHora().format(DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm:ss"))
```

Esto pondrá nuestra fecha y hora en el formato que espera la tabla. Fijaos que en la fecha primero está el año y no el día.

Por último, tener en cuenta que las operaciones que devuelven una tabla (las de lectura) se ejecutan en el `Statement` con un `executeQuery` y devuelven un `ResultSet`. El resto de operaciones se ejecutan con `execute` y no devuelven nada (al menos por ahora).

Programa principal

Hora de comprobar los frutos de nuestro trabajo. Vamos a necesitar el archivo sql que creará la tabla y la poblará de datos si así lo deseamos, que está descargable en los recursos de la actividad.

Lo primero de todo, vamos a preguntar al usuario si quiere volver a crear la tabla de facturas. Lo haremos ejecutando un scripts que eliminará la tabla si existe y volverá a crearla y a poblarla con nuestros datos ya que hemos activado la opción de ejecutar órdenes múltiples en nuestra conexión. Por lo tanto, preguntaremos al usuario y en función de su respuesta introducida por teclado ejecutaremos o no el script. Usaremos para ello un método que crearemos más adelante.

```
import ...

public class Main {

    private static final Logger logger = LogManager.getLogger(Main.class.getName());  13 usages

    public static void main (String[] args) {
        System.out.println("----------------------------------------");
        System.out.println("----------------------------------------");
        System.out.println("---------- UT2 A5 - CRUD Básico ---------");
        System.out.println("----------------------------------------");
        System.out.println("----------------------------------------");

        Scanner sc = new Scanner(System.in);
        System.out.println("¿Quieres crear la tabla de Facturas?");
        System.out.println("1-Sí");
        System.out.println("Cualquier otra cosa - No");
        String respuesta = sc.nextLine();
        if (respuesta.equals("1")) {
            try {
                ejecutarSQL( fichero: "src/main/resources/SQL/facturas.sql");
            } catch (IOException e) {
                logger.error( message: "Error al leer el fichero: ", e.fillInStackTrace());
                System.exit( status: -1);
            } catch (SQLException e) {
                logger.error( message: "Error al ejecutar el fichero de creación de Facturas: ", e.fillInStackTrace());
                System.exit( status: -1);
            }
        }
```

Seguidamente consultaremos con el usuario si quiere probar el CRUD. En caso de respuesta positiva haremos una serie de pruebas con cada uno de nuestros métodos de clase DAO antes de terminar el programa. Con respuesta negativa, acabamos el programa directamente.

```
    System.out.println("¿Quieres probar el CRUD de Facturas?");
    System.out.println("1-Sí");
    System.out.println("Cualquier otra cosa - No");
    respuesta = sc.nextLine();
    if (respuesta.equals("1"))
        testCRUD();
}

private static void testCRUD() {  1 usage
    FacturaDAO facturaDAO = new FacturaDAO();
    insertarFactura(facturaDAO);
    consultarTodasLasFacturas(facturaDAO);
    actualizarFactura(facturaDAO);
    consultarFactura(facturaDAO);
    eliminarFactura(facturaDAO);
}
```

Creamos un método para probar cada uno de los DAO y los llamamos secuencialmente para hacer las pruebas. **Accedemos a la base de datos exclusivamente a través de los métodos DAO.**

Métodos auxiliares

Toca bajar al barro. Vamos a escribir los métodos usados en el programa principal. Primero de todo los que nos permiten volver a crear la tabla `Facturas` a partir de un script. Seguidamente crearemos los métodos que nos sirven para probar los métodos CRUD a través del DAO.

Leer fichero SQL

Leemos el fichero SQL línea a línea y lo guardaremos en un `String`. Recuerda que al leer con `BufferedReader` los saltos de línea desaparecen, así que tendremos que añadirlos nosotros tras leer cada una de ellas. Usaremos `System.lineSeparator()` que es independiente del sistema operativo.

```
private static String leerFicheroSQL(String sqlFilePath) throws IOException {
    // Leer el archivo SQL y construir el script en un StringBuilder
    StringBuilder script = new StringBuilder();
    try (BufferedReader br = new BufferedReader(new FileReader(sqlFilePath))) {
        String line;
        while ((line = br.readLine()) != null) {
            // Añadir cada línea al script con el separador
            //      de línea del sistema (independiente de Windows o Linux)
            script.append(line).append(System.lineSeparator());
        }
    }

    return script.toString();
}
```

Ejecutar fichero SQL

Ahora hacemos uso del método anterior para almacenar en un `String` el contenido de nuestro archivo SQL. Seguidamente los ejecutaremos.

Para ello, primero estableceremos una conexión con la base de datos a través de nuestra clase de conexión. Seguidamente, ejecuta las órdenes SQL de manera similar a lo visto en la presentación teórica. Recuerda hacer buen uso del *logger* para dar cuenta del éxito o fracaso de nuestra operación.

```
private static void ejecutarSQL(String fichero) throws IOException, SQLException {
    String contenidoFicheroSQL = leerFicheroSQL(fichero);

    Connection connection = MySQLConnection.getInstance().getConnection();
    if (connection != null) {
        // Ejecutar todas las consultas en un único executeUpdate()
```

3. Completa el resto del método *ejecutarSQL(String)* que creará y poblará la tabla *Facturas*

Métodos CRUD

Vamos a crear un pequeño test de cada uno de los métodos CRUD. En aquellos que necesiten un código para buscar el registro apropiado lo introduciremos directamente *hardcoded* ya que simplemente estamos probando los métodos.

```
private static void insertarFactura(FacturaDAO facturaDAO) {  1 usage
    System.out.println("\n\n\n----------------------------------------");
    System.out.println("   1 - Create - Insertar una Factura   ");
    System.out.println("----------------------------------------");
```

```
private static void consultarFactura(FacturaDAO facturaDAO) {  1 usage
    System.out.println("\n\n\n----------------------------------------");
    System.out.println("   2 - Read - Consultar una Factura   ");
    System.out.println("----------------------------------------");
```

```
private static void consultarTodasLasFacturas(FacturaDAO facturaDAO) {  1 usage
    System.out.println("\n\n\n------------------------------------------------");
    System.out.println("   3 - Read - Consultar todas las Facturas   ");
    System.out.println("------------------------------------------------");
```

```
private static void actualizarFactura(FacturaDAO facturaDAO) {  1 usage
    System.out.println("\n\n\n----------------------------------------");
    System.out.println("   4 - Update - Modificar Factura   ");
    System.out.println("----------------------------------------");
```

Por ejemplo, el método que nos permite eliminar un registro sería algo similar a lo siguiente. Usaremos el código de un registro de los que pueblan la base de datos.

```
private static void eliminarFactura(FacturaDAO facturaDAO) {  1 usage
    System.out.println("\n\n\n--------------------------------------");
    System.out.println("   5 - Delete - Eliminar Factura   ");
    System.out.println("--------------------------------------");

    try {
        facturaDAO.eliminar( codigo: 27792);
        // IMPORTANTE: si lo ejecutáis varias veces sin volver a crear la tabla fallará
        logger.info("(Delete) Has eliminado a Mark Zuckerberg, código 27792, con éxito.");
    } catch (SQLException e) {
        logger.error( message: "Error al eliminar la factura con código 27792. " +
                "Mark Zuckerberg es más resistente de lo esperado:\n", e.fillInStackTrace());
    }
}
```

4. Completa el resto de los métodos de prueba *insertarFactura(FacturaDAO), consultarTodasLasFacturas(FacturaDAO), actualizarFacturas(FacturasDAO) y consultarFacturas(FacturasDAO).* Para el método *actualizarFacturas()* primero deberás recuperar una factura por su código y seguidamente multiplicar el importe de la factura por 100000, antes de volver a guardarla en la base de datos.

Y con esto ya tenemos el programa preparado para ejecutarlo.

Ampliación:

Revisa las clases que manejan fechas y horas y los métodos que nos permiten formatearlas