

TEMA

Tema 2. Persistencia en BDR con JDBC

Desarrollo de aplicaciones
multiplataforma

Acceso a datos

Autora: Silvia Macho



Tema 2: Persistencia en BDR con JDBC

¿Qué aprenderás?

- Crear una aplicación en Java que utilice una BBDD como elemento de persistencia.
- Utilizar la API JDBC para integrarla en una aplicación Java.

¿Sabías que...?

- Los SGBD son un elemento fundamental en una aplicación ya que se encarga de la persistencia de los datos.
- El desfase Objeto-Relacional es un elemento muy a tener en cuenta debido a que las BBDD tradicionalmente son relacionales y los lenguajes de programación actuales cada vez tienden más a estar orientados a objetos.



2.1. Introducción

Un SGBD es una aplicación especializada en el almacenamiento de datos estructurados, con la capacidad de guardarlos y recuperarlos de forma consistente y con gran eficacia, independientemente del número de accesos que se realicen de forma simultánea.

El desarrollo de aplicaciones utiliza las bases de datos como herramienta de almacenamiento. Para conseguir coordinar los diferentes lenguajes de programación con la potencia de los distintos SGBD, se han desarrollado un conjunto de herramientas específicas para conseguir conectar una aplicación con una determinada base de datos e enviarle la secuencia de tareas que las aplicaciones puedan llegar a necesitar durante su ejecución. Estas herramientas son los llamados conectores.

Un conector es un conjunto de clases encargadas de implementar la API con un lenguaje de programación, para facilitar el acceso a una base de datos. Para que una aplicación pueda conectarse a una base de datos y ejecutar sentencias en ella, necesita un conector asociado.

Cada SGBD tiene su propio lenguaje para gestionar sus datos mediante sentencias de consulta y manipulación, pero cuando se quiere acceder desde una aplicación hecha con un determinado lenguaje de programación independientemente del sistema gestor que contenga esos datos, la aplicación tiene que hacer uso de conectores. Un conector da a la aplicación una forma homogénea de acceder a cualquier SGBD.



2.2. Desfase Objeto-Relacional

El problema del desfase objeto-relacional aparece como consecuencia de tener aplicaciones programadas con un lenguaje orientado a objetos que acceden a bases de datos relacionales.

Esta situación ocurre cuando:

- Implementamos una aplicación con un lenguaje de programación orientado a objetos y el lenguaje de acceso a los datos no sigue ese mismo paradigma de programación.
- Especificamos los tipos de datos. En las bases de datos relacionales, solemos utilizar tipos de datos simples, mientras que en la programación orientada a objetos, utilizamos datos complejos.
- Cuando desarrollamos una aplicación, realizamos una traducción del modelo de objetos al modelo entidad-relación, ya que la aplicación trabaja con objetos y la base de datos con tablas, cosa que hace que como desarrolladores tengamos que hacer dos diagramas para el desarrollo completo de la aplicación.

El modelo relacional es una forma de organizar los datos de una aplicación agrupándolos según la relación que podamos establecer entre ellos de acuerdo con el modelo. Esta agrupación se materializa en forma de tabla, donde las columnas representan en conjunto de datos i las filas son los propios datos de la aplicación.

El modelo de objetos se basa en organizar los datos en clases, objetos y las relaciones entre ellos. Cada objeto pertenece a una clase, que le da su estructura y funcionalidad.

Todas estas diferencias que hemos comentado, constituyen lo que conocemos como desfase objeto-relacional. Este desfase nos obliga a utilizar herramientas extra si decidimos trabajar con estos dos mundos de forma conjunta. Estas herramientas son los conectores.



2.2.1. Conectores

Los SGBD tienen sus propios protocolos de comunicación que facilitan su acceso, lo que hace que los programadores tengamos que aprender un lenguaje nuevo para poder trabajar con cada sistema gestor. Podemos reducir ese número de protocolos y lenguajes, utilizando una interfaz que nos proporciona métodos para acceder a la base de datos.

Estas interfaces proporcionan métodos para:

- Crear una conexión a la base de datos.
- Ejecutar sentencias de consulta y manipulación sobre la base de datos.
- Procesar los resultados obtenidos de la ejecución de una consulta sobre la base de datos.

Las diferentes opciones de las que disponemos, extraen la complejidad de cada sistema y proporcionan una interfaz común, basada en SQL, para poder acceder de forma homogénea a los datos. Un ejemplo es JDBC.

Al conjunto de clases encargadas de implementar una API y con ello facilitar el acceso a una base de datos, se llama conector y driver. Para poder conectarnos a una base de datos y poder ejecutar sentencias (ya sean de consulta o manipulación de datos), toda aplicación necesita tener un conector asociado.

Cuando implementamos una aplicación con base de datos, el conector oculta los detalles específicos de cada BBDD, así como programadores, solo nos tenemos que preocupar de los aspectos propios de la aplicación, dejando a un lado otros aspectos. La mayoría de los fabricantes, ofrecen conectores para acceder a sus bases de datos.



2.3. La API JDBC

En 1997 la empresa Sun Microsystems creó JDBC (Java Database Connectivity), una API que conecta base de datos implementada específicamente para el lenguaje Java. La funcionalidad está encapsulada en clases, ya que Java es un lenguaje completamente orientado a objetos, y no dependen de una plataforma específica, cumpliendo con las características propias de Java.

Sun Microsystems creó una API que contiene un conector específico de Java que puede comunicarse directamente con cualquier base de datos utilizando controladores (drivers), además de incorporar un driver especial que actúa de adaptador entre JDBC y ODBC, que es el estándar de acceso a bases de datos. Este driver se llama bridge JDBC-ODBC, permitiendo que una aplicación Java se pueda conectar a cualquier base de datos que cumpla con el estándar.

Actualmente, la mayoría de los SGBD disponen de drivers JDBC, pero en el caso de no disponer, siempre que el sistema implemente el estándar, se podrá trabajar con él a través del bridge JDBC-ODBC.

El driver JDBC actúa como una capa de software intermedia, situada entre el programa en Java y el SGBD que utiliza SQL. Esta capa es independiente de la plataforma y del SGBD utilizado.

Con el conector JDBC no tenemos que escribir un programa diferente en función del SGBD que estemos utilizando, al contrario, escribiremos una única aplicación utilizando la API JDBC, y la API se encargará de enviar las sentencias a la BBDD utilizada en cada situación.



2.3.1. Tipos de conectores JDBC

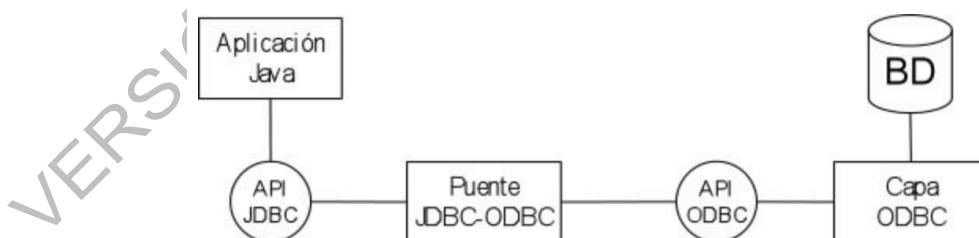
A conector JDBC está formado por los siguientes elementos:

- API JDBC: permite el acceso desde Java a BBDD relacionales y la ejecución de sentencias, tanto de consulta como de manipulación de datos.
- El gestor del conector JDBC (DriverManager): es el elemento encargado de conectar la aplicación Java con el driver JDBC que corresponda.
- El paquete de pruebas: permite comprobar si un conector o driver cumple con las especificaciones de JDBC.
- El brigde JDBC-ODBC: permite utilizar los drivers estándar ODBC como si fueran de tipo JDBC.

A partir de los elementos anteriores, existen 4 tipos de controladores JDBC. Estos controladores se numeran de I a IV, y el número depende del nivel de independencia respecto a la plataforma.

- Tipo I, Puente JDBC-ODBC. ODBC:

Fue creado para proporcionar una conexión a bases de datos en Microsoft Windows. El puente JDBC-ODBC permite enlazar Java con cualquier base de datos disponible en ODBC. Cada cliente debe tener instalado el driver.

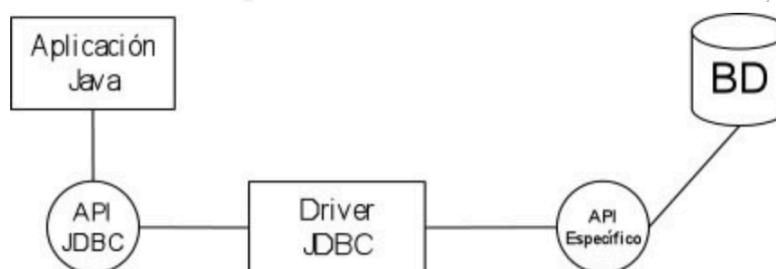


Fuente: <http://www.jtech.ua.es/j2ee/publico/ja-2012-13/sesion07-apuntes.html>



- **Tipo II, Parte Java, parte driver nativo:**

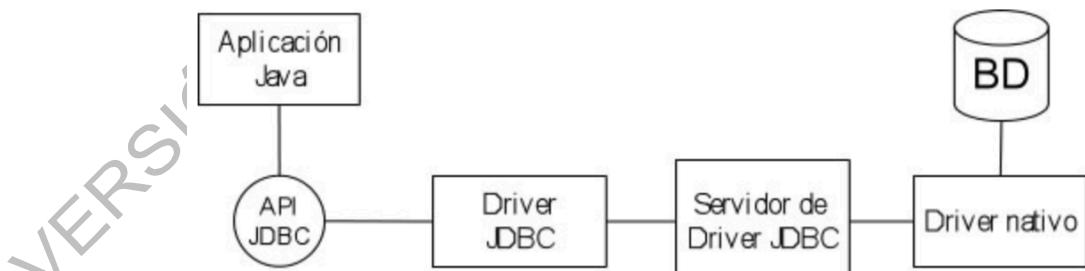
Es una combinación de implementación Java y API nativo para el acceso a la base de datos. Este tipo de driver es más rápido que el anterior, pues no se realiza el paso por la capa ODBC. Las llamadas JDBC se traducen en llamadas específicas del API de la base de datos. Cada cliente debe tener instalado el driver. Tiene menor rendimiento que los dos siguientes y no se pueden usar en Internet, ya que necesita el API de forma local.



Fuente: <http://www.jtech.ua.es/j2ee/publico/lja-2012-13/sesion07-apuntes.html>

- **Tipo III, Servidor intermediario de acceso a base de datos:**

Este tipo de driver proporciona una abstracción de la conexión. El cliente se conecta a los SGBD mediante un componente servidor intermedio, que actúa como una puerta para múltiples servidores. La ventaja de este tipo de driver es el nivel de abstracción.

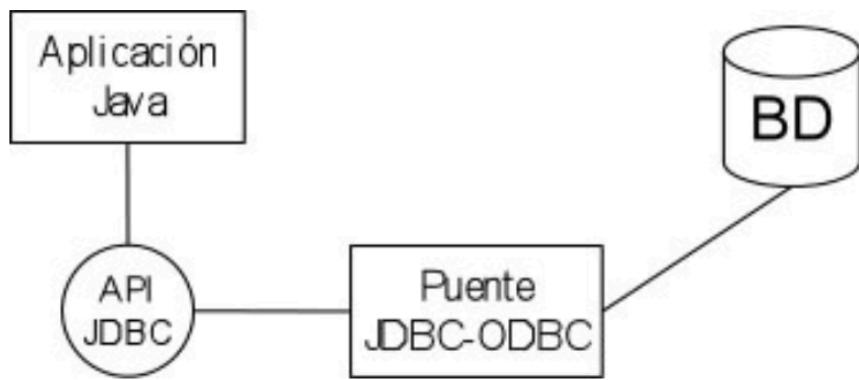


Fuente: <http://www.jtech.ua.es/j2ee/publico/lja-2012-13/sesion07-apuntes.html>



- Tipo IV, Drivers Java.

Este es el más directo. La llamada JDBC se traduce directamente en una llamada de red a la base de datos, sin intermediarios. Proporcionan mejor rendimiento. La mayoría de SGBD proporcionan drivers de este tipo.

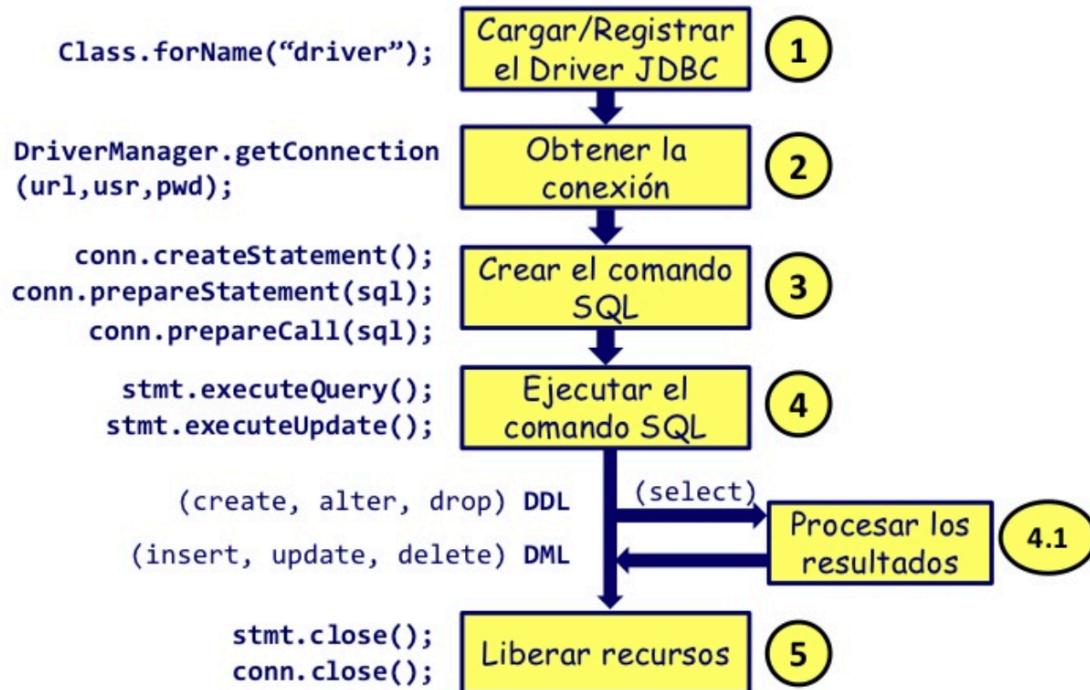


Fuente: <http://www.jtech.ua.es/j2ee/publico/ija-2012-13/sesion07-apuntes.html>



2.4. Acceso a BBDD utilizando JDBC

Para acceder a una BBDD con JDBC, tenemos que seguir los siguientes pasos:



Fuente: <https://es.slideshare.net/solokuko/curso-basico-de-base-de-datos-con-java>

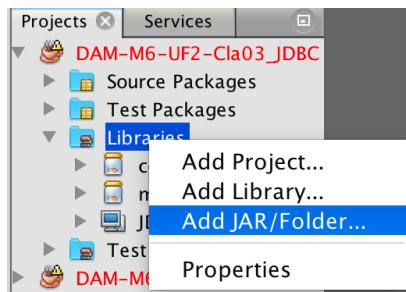
Lo primero que necesitamos para acceder a una base de datos, es un objeto conector. Este objeto, es el que sabe cómo conectarse e interactuar con la base de datos. Java no tiene todos los conectores de todas las bases de datos existentes en el mercado, por lo que tenemos que añadir el conector correspondiente en cada caso.

Por ejemplo, para acceder a una BBDD MySQL, utilizaremos el connector o driver `mysql-connector-java.5.1.21-bin.jar` (la numeración representa la versión del driver, cosa que puede cambiar). Tendremos que añadir este archivo al proyecto en Java que estemos implementando.

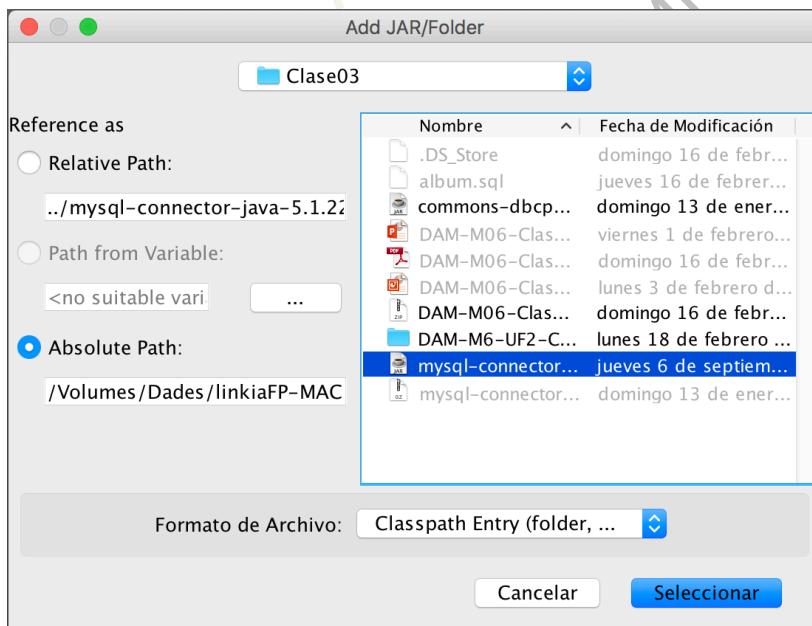


En el caso de utilizar NetBeans, una forma rápida de añadir este driver es:

1. Crear el nuevo proyecto que representa la aplicación.
2. En la ventana de proyectos, pestaña Libraries, hacemos click con el botón derecho y seleccionamos la opción “Add JAR/Folder”:

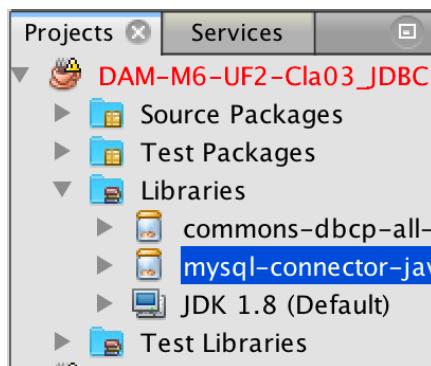


3. Desde esta opción, se abrirá una nueva ventana que nos dejará explorar nuestro equipo hasta la ubicación del archivo que buscamos.





4. Una vez hayamos seleccionado el archivo, ya formará parte de la aplicación y podremos utilizar el driver para acceder a la base de datos:



Una vez hemos añadido el driver a la aplicación, ya lo podremos cargar desde la aplicación para iniciar el proceso de conexión y acceso a la base de datos.

2.4.1. La API JDBC

La interfaz del conector JDBC está en los paquetes `java.sql` y `javax.sql`. El contenido de estos dos paquetes son en su mayoría interfaces, ya que la implementación en clases es responsabilidad de cada proveedor de BBDD para poderse adaptar al protocolo correspondiente.

Independientemente de la implementación de cada proveedor, las clases e interfaces básicas para trabajar con JDBC son las siguientes:

- La clase `Connection`. Los objetos de esta clase representan una conexión a la base de datos. A través de esta conexión, la aplicación Java podrá leer y escribir datos, así como explorar la estructura de la base de datos.
Estos objetos los creamos con la llamada `DriverManager.getConnection()` o `DataSource.getConnection()`.
- La interfaz `DriverManager`. Esta interfaz es complementaria a `Connection`. Con esta interfaz podemos registrar los controladores JDBC y proporcionar las conexiones a la base de datos.



- La clase Statement. Esta clase proporciona los métodos para que las sentencias, utilizando el lenguaje SQL, las podamos ejecutar sobre la base de datos. En el caso que la sentencia sea de tipo consulta, los métodos de esta clase nos permiten recuperar el resultado (a través de la clase ResultSet).
- La clase SQLException. Esta clase representa una excepción generada como resultado de la ejecución de un método sobre la base de datos, como por ejemplo errores en la conexión, sentencias SQL incorrectas, falta de privilegios, etc. Esto es lo que hace que todas las líneas de código Java que utilicen la API JDBC tienen que ir dentro de un bloque try-catch.

Además de estas clases/interfaces básicas, la API JDBC proporciona la posibilidad de acceder a los metadatos de la BBDD. Estos metadatos nos dan información sobre la estructura de la BBDD y gracias a esta información, podemos desarrollar aplicaciones independientes del esquema que tenga la BBDD. Las clases asociadas a esta funcionalidad son:

- La interfaz DatabaseMetaData. Los objetos de esta clase proporcionan la posibilidad de trabajar con la estructura y capacidades de la BBDD. Se crean llamando al método connection.getMetaData(). Los metadados son datos acerca de los datos, es decir, datos que informan de la naturaleza de otros datos. Esta interfaz contiene métodos para recuperar la información de una BBDD, así como información del driver JDBC que esté utilizando. Estos métodos son útiles cuando creamos aplicaciones genéricas que pueden acceder a diferentes bases de datos.
- La clase ResultSetMetaData. Los objetos de esta clase son el resultado de ejecutar una consulta sobre un objeto DatabaseMetaData. Los métodos de esta clase permiten determinar las características de un objeto ResultSet, es decir, permite determinar, por ejemplo, el número de columnas de un tabla, información sobre las columnas, etc.



2.4.2. Conexión a una BBDD con JDBC

El siguiente código de ejemplo, accede a una base de datos llamada discográfica, creada en MySQL. Esta base de datos tiene una tabla Album. La tabla contiene tres campos:

- ID: clave primaria numérica
- Título: varchar de 20
- Autor: varchar de 20

#	Nombre	Tipo	Cotejamiento	A1
1	id	int(11)		
2	título	varchar(20)	latin1_swedish_ci	
3	autor	varchar(20)	latin1_swedish_ci	

El código de conexión a la base de datos desde la aplicación Java, es el siguiente:

```
Connection conn1 = null;

public Gestor_conexion() {
    // ABRE UNA CONEXIÓN A UNA BASE DE DATOS QUE SE SUPONE MySQL Y QUE TIENE LAS TABLAS
    // Y LOS USUARIOS CREADOS SEGÚN ESTE EJEMPLO.
    try {
        //RECUEDE: PARA EJECUTAR ESTE CÓDIGO ES NECESARIO TENER MySQL FUNCIONANDO
        //Y LAS TABLAS Y USUARIOS CREADOS
        String url1 = "jdbc:mysql://localhost:3306/discografica";
        String user = "root";
        String password = "";
        conn1 = DriverManager.getConnection(url1, user, password);
        if (conn1 != null) {
            System.out.println("Conectado discografica");
        }
    } catch (SQLException ex) {
        System.out.println("ERROR:La dirección no es válida o el usuario y clave");
        ex.printStackTrace();
    }
}
```

Los pasos para obtener una conexión a una base de datos utilizando JDBC son muy parecidos, independientemente del tipo de BBDD con el que vayamos a trabajar.



El primer paso es cargar al conector. Todos los conectores o drivers JDBC, tienen que implementar la interfaz `java.sql.Driver`. La carga de este driver, originariamente, la realizábamos con `Class.forName(driver)`. Actualmente, al poner el archivo que representa al driver en el propio proyecto (en la carpeta `lib`), cuando la clase `DriverManager` se inicializa, busca el archivo en el proyecto y detecta el driver que corresponda.

Una vez cargado el driver, podemos crear una conexión. El resultado es un objeto de la clase `java.sql.Connection`. Para conseguirlo, utilizaremos el método `DriverManager.getConnection(url)`. En el ejemplo, la línea `String url1 = "jdbc:mysql://localhost:3306/discografica";` representa el valor `String` que le pasamos al método `getConnection` para establecer la conexión. Este `String`, indica que vamos a acceder a una BBDD MySQL a través de JDBC, que la BBDD está ubicada en `localhost` (`127.0.0.1`), que para conectarnos utilizaremos el puerto `3306` y que el nombre de la BBDD es `discografica`. El método `getConnection` tiene dos parámetros más, que nos permiten especificar el usuario y contraseña que usaremos para acceder a la BBDD.

Si todo funciona correctamente, la ejecución del método devolverá un objeto de la clase `Connection` que representará la conexión a la BBDD y que utilizaremos en el resto de la aplicación para acceder a la base de datos. En el caso que fuera mal, como el código está dentro de un bloque `try-catch`, la cláusula `catch` capturaría la excepción generada a partir del error en la ejecución del método.



Cuando ya no queramos acceder a la base de datos, una acción que debemos realizar, es cerrar esta conexión. Para hacerlo, tenemos el método `close()`. Aquí hay un pequeño ejemplo:

```
public void cerrar_Conexion() {  
    //SE CIERRA LA CONEXIÓN  
    try {  
        conn1.close();  
    } catch (SQLException ex) {  
        System.out.println("ERROR: al cerrar la conexión");  
        ex.printStackTrace();  
    }  
}
```

2.4.3. Sentencias de definición de datos con JDBC

El lenguaje de definición de datos (DDL: Data Definition Language), se encarga de definir la BBDD. Consta de sentencias para crear la estructura de la base de datos y definir gran parte de su nivel interno.

Las principales sentencias del lenguaje DDL siempre se utilizan junto con el tipo y el nombre del objeto. Estas sentencias son:

- **CREATE:** permite crear una base de datos o un objeto.
- **ALTER:** permite modificar la estructura de una base de datos o de un objeto.
- **DROP:** permite eliminar una base de datos o un objeto.



Para ejecutar una sentencia SQL en una base de datos utilizando JDBC, utilizamos un objeto de la clase Statement. Este objeto lo obtenemos a partir del objeto Connection que representa la conexión a la base de datos de la siguiente forma:

```
Statement st = connection.createStatement();
```

La clase Statement tiene muchos métodos, pero los más utilizados son:

- executeQuery: método que permite ejecutar consultas, es decir sentencias de tipo SELECT.
- executeUpdate(): método que permite ejecutar sentencias de manipulación de datos (INSERT, UPDATE, DELETE ...).

El siguiente código de ejemplo, modifica la estructura de una tabla desde la aplicación Java:

```
Statement st = conn1.createStatement();
int contador = st.executeUpdate("ALTER TABLE Album ADD pistas");
```

Este otro ejemplo, elimina la tabla Album de la base de datos:

```
st.executeUpdate("DROP TABLE Album");
```



2.4.4. Sentencias de manipulación de datos con JDBC

Las sentencias de manipulación de datos (DML: Data Manipulation Language), son las que permiten insertar, modificar, borrar y consultar los datos que hay en la BBDD. Las sentencias DML son:

- SELECT: permite la consulta de información en una BBDD y la selección de una o más filas y columnas de una o más tablas.
- INSERT: permite añadir datos a una tabla.
- UPDATE: permite modificar los datos de una tabla.
- DELETE: permite eliminar datos de una tabla.

El siguiente código de ejemplo, inserta valores en una tabla desde la aplicación Java:

```
public void Insertar() {  
    try {  
        // CREA UN STATEMENT PARA UNA CONSULTA SQL INSERT.  
        Statement sta = conn1.createStatement();  
        sta.executeUpdate("INSERT INTO album VALUES (3, 'Black Album', 'Metallica')");  
  
        int id = 8;  
        String tit = "Songs of experience";  
        String aut = "U2";  
        sta.executeUpdate("INSERT INTO album VALUES (" + id + ", '" + tit + "', '" + aut + "')");  
  
        sta.close();  
    } catch (SQLException ex) {  
        System.out.println("ERROR: al hacer un Insert");  
        ex.printStackTrace();  
    }  
}
```

En el ejemplo, ejecutamos dos sentencias de inserción. La primera crea la sentencia a ejecutar como un String con todos los datos que necesita la sentencia para poder ejecutarse correctamente. La segunda, crea la sentencia a ejecutar concatenando los valores almacenados en variables, cosa que nos permite ver que los valores que le pasamos a una sentencia, pueden ser fijos o los podemos pasar como parámetros.

Al finalizar la ejecución de las sentencias, cerramos la conexión utilizando el método close().



2.4.5. Sentencias de consulta de datos con JDBC

Para la ejecución de consultas sobre una BBDD desde una aplicación en Java, utilizaremos las clases Statement y ResultSet, junto con los métodos createStatement() y executeQuery(). Cuando hablamos de sentencias de tipo consulta, nos referimos a la sentencia SELECT.

El método executeQuery de la clase Statement recibe como parámetro de tipo String la sentencia a ejecutar. El resultado de la ejecución de la consulta, estará almacenada en un objeto de la clase ResultSet. Esta clase es un conjunto ordenado de las filas de una tabla. En la clase ResultSet, tenemos los métodos next() y getXXX(), que nos permiten iterar por las filas y obtener los valores de los campos que necesitemos para nuestra aplicación. Al llamar al método next(), tendremos disponible el resultado de la consulta almacenado en el objeto ResultSet. La forma de recoger los valores de los campos, es utilizar los métodos getXXX(). Existe un método getXXX para cada uno de los tipos de datos, es decir, getInt(), getString(), etc, de tal forma, que si sabemos qué tipo de datos hay en una determinada columna, recoger su valor es tan sencillo como llamar al método getXXX() que se corresponda con su tipo de datos. Si no tenemos esta información, siempre podremos utilizar el método getObject(), que es capaz de recuperar cualquier tipo de datos.



El siguiente ejemplo, ejecuta una consulta sobre la tabla para recuperar todos sus registros:

```
public void Consulta_Statement() {
    //CRAR UN ESTATEMENT PARA UNA CONSULTA SELECT
    try {
        Statement stmt = conn1.createStatement();
        String query = "SELECT * FROM album";
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            System.out.println("ID - " + rs.getInt("id")
                + ", Título " + rs.getString("titulo")
                + ", Autor " + rs.getString("autor"));
        }
        rs.close();
        stmt.close();
    } catch (SQLException ex) {
        System.out.println("ERROR:al consultar");
        ex.printStackTrace();
    }
}
```

En el siguiente ejemplo, también hacemos una consulta en una tabla, pero en este caso, limitamos más el tipo de registros que estamos buscando (todos aquellos cuyo título contiene la B):

```
public void Consulta_Statement() {
    //CRAR UN ESTATEMENT PARA UNA CONSULTA SELECT
    try {
        Statement stmt = conn1.createStatement();
        String query = "SELECT * FROM album WHERE titulo like '%B%'";
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            System.out.println("ID - " + rs.getInt("id")
                + ", Título " + rs.getString("titulo")
                + ", Autor " + rs.getString("autor"));
        }
        rs.close();
        stmt.close();
    } catch (SQLException ex) {
        System.out.println("ERROR:al consultar");
        ex.printStackTrace();
    }
}
```



En ambos ejemplos, el resultado de la consulta se guarda en un objeto de tipo ResultSet. Con el método next() en un bucle while, iteramos por todos y cada uno de los registros que tenemos en el objeto como resultado de la consulta. En cada vuelta del bucle procesaremos un registro. Este proceso consiste en obtener cada una de las columnas con su correspondiente método getXXX() en función del tipo de datos y después crear un String formado por la concatenación de estos valores, para mostrarlos finalmente por pantalla.

2.4.6. Sentencias precompiladas con JDBC

La clase Statement es una clase madre de una serie de clases que permiten crear sentencias de diferentes tipos que después podemos ejecutar sobre una BBDD desde una aplicación Java.

Una primera clase hija es PreparedStatement. Esta clase nos permite ejecutar sentencias SQL pre-compiladas, es decir, permite que los valores de ejecución de la sentencia sean establecidos de forma dinámica. La diferencia fundamental con Statement es que las sentencias pueden tener valores indefinidos, que estableceremos utilizando el símbolo de interrogación (?). En las sentencias, pondremos tantos interrogantes como parámetros queramos pasar a la sentencia.

Antes de poder ejecutar la sentencia, tendremos que dar valor a cada uno de los parámetros, es decir, a cada uno de los interrogantes. Para ello, tenemos los métodos setXXX() donde XXX se corresponde con el tipo de datos del parámetro al que le queremos dar el valor. Estos métodos reciben dos parámetros, el primero es la posición que ocupa el parámetro o interrogante al que le queremos dar el valor y el segundo es el valor que queremos asignar.



Esta clase la podemos utilizar para cualquier tipo de sentencias, tanto para las de consulta como para las de manipulación de datos.

El siguiente ejemplo ejecuta una consulta, pero parametrizando el criterio de búsqueda:

```
public void Consulta_preparedStatement() {
    //CREAR UN STATEMENT PARA UNA CONSULTA SELECT CON PARÁMETROS
    try {
        String query = "SELECT * FROM album WHERE titulo like ?";
        PreparedStatement pst = conn1.prepareStatement(query);
        pst.setString(1, "%B%");

        ResultSet rs = pst.executeQuery();
        while (rs.next()) {
            System.out.println("ID - " + rs.getInt("id")
                + ", Título " + rs.getString("titulo")
                + ", Autor " + rs.getString("autor"));
        }
        rs.close();
        pst.close();
    } catch (SQLException ex) {
        System.out.println("ERROR:al consultar");
        ex.printStackTrace();
    }
}
```

Este otro ejemplo, realiza una inserción de datos, parametrizando la sentencia:

```
public void Insertar_preparedStatement() {
    try {

        // CREA UN STATEMENT PARA UNA CONSULTA SQL INSERT CON PARÁMETROS
        String query = "INSERT INTO album " + "VALUES (?, ?, ?)";
        PreparedStatement pst = conn1.prepareStatement(query);
        pst.setInt(1, 8);
        pst.setString(2, "Black Album");
        pst.setString(3, "Metallica");

        pst.executeUpdate();
        pst.close();
    } catch (SQLException ex) {
        System.out.println("ERROR:al hacer un Insert");
        ex.printStackTrace();
    }
}
```



2.4.7. Procedimientos almacenados con JDBC

Otra clase hija de Statement, es CallableStatement. Esta clase nos permite crear sentencias de tipo preparedStatement que llaman a procedimientos almacenados en la base de datos, es decir, métodos incluidos dentro de la propia BBDD. No todos los SGBD trabajan con este tipo de procedimientos. La forma de trabajar con objetos de esta clase, es la misma que hemos visto para la clase preparedStatement, aunque en este caso, el valor que de damos a cada uno de los parámetros se corresponden con todos y cada uno de los parámetros que tiene definido el procedimiento almacenado.

En el siguiente ejemplo, llamamos al método almacenado dameAlbum(título, autor), que nos devolverá todos los registros de la tabla Album que coincidan con el título y autor pasados como parámetro.

```
CallableStatement cs = conn1.prepareCall("CALL dameAlbum(?,?)");  
// Proporcionamos los valores para el procedimiento  
cs.setString(1, "R%");  
cs.setString(2, "Pablo Alboran");
```



2.5. Gestión de transacciones con JDBC

Una de las características de la mayoría de SGBD es la gestión de transacciones. Una transacción es un conjunto de sentencias que se ejecutan como una única unidad, es decir, de forma indivisible. Una transacción se inicia cuando la aplicación se encuentra con una sentencia DML y finaliza cuando se encuentra una de las siguientes sentencias:

- Un COMMIT o ROLLBACK.
- Una sentencia DDL, como por ejemplo CREATE.
- Una sentencia DCL, es decir, aquellas sentencias que permiten al administrador controlar el acceso a la BBDD, como por ejemplo GRANT o REVOKE.

Una transacción que se ejecuta correctamente (todas las sentencias que la componen), se confirma con la sentencia COMMIT. En cambio, cuando una de las sentencias que forma parte de la transacción no se ejecuta correctamente, la transacción se cancelará utilizando la sentencia ROLLBACK. En el momento que una transacción se cancela, se deshacen todas las sentencias que forman la transacción que se hayan podido ejecutar. Dicho de otra forma, o se ejecutar todas las sentencias que forman la transacción o ninguna, no hay término medio.

EN JDBC por defecto cada sentencia SQL se confirma en el momento que se ejecuta, por lo que inicialmente cada sentencia por si misma es una transacción, es decir, tiene un funcionamiento conocido como auto-commit. Para poder agrupar sentencias en transacciones, tenemos que modificar este comportamiento y deshabilitar este modo auto-commit que viene por defecto. Para hacerlo, tenemos el método setAutocommit(). Este método tiene como parámetro un valor booleano que será el que nos permita activar/desactivar el modo auto-commit.

Además de este método para modificar ese comportamiento por defecto, también tenemos los métodos commit() y rollback() que permiten confirmar o cancelar una transacción respectivamente.



En el siguiente ejemplo, ejecutamos dos sentencias de inserción en la BBDD con el uso de transacciones:

```
public void Insertar_con_commit() {
    //INSERTA CON UN SQL INSERT PERO RESPETANDO TRANSACCIONES
    try {
        conn1.setAutoCommit(false);
        // create our java jdbc statement
        Statement sta = conn1.createStatement();
        sta.executeUpdate("INSERT INTO album " + "VALUES (3, 'Black Album', 'Metallica')");
        sta.executeUpdate("INSERT INTO album " + "VALUES (9, 'A kind of magic', 'Queen')");
        conn1.commit();
    } catch (SQLException ex) {
        System.out.println("ERROR: al hacer un Insert");
        try {
            if (conn1 != null) {
                conn1.rollback();
            }
        } catch (SQLException se2) {
            se2.printStackTrace();
        }//end try
        ex.printStackTrace();
    }
}
```

Si no fijamos en el ejemplo, lo primero que hacemos es deshabilitar el auto-commit. A partir de ese momento las sentencias serán tratadas como parte de una misma transacción. Estas sentencias se confirman con la llamada conn1.commit(). Si alguna de estas sentencias no se ejecutara correctamente, generaría una excepción y por ello la llamada al rollback, que es el que se encarga de cancelar la transacción, lo hemos puesto en la cláusula catch, para que en el momento que se genere dicha excepción, se cancelen todas las sentencias y por lo tanto la transacción.



2.6. Pool de conexiones con JDBC

Para aplicaciones sencillas, la forma de obtener el objeto de la clase Connection que nos representa la conexión a la base de datos ya es suficiente ya que solo necesitamos una única conexión. Para aplicaciones más complejas, donde utilizamos varios hilos de ejecución, éstos no pueden utilizar la misma conexión con la BBDD de forma simultánea, ya que la información de entrada/salida se mezclaría entre los diferentes hilos.

Soluciones que podemos encontrar a este problema:

- Abrir y cerrar una conexión cada vez que la necesitemos. Con esta solución, cada hilo de ejecución tendrá su propia conexión y nos ahorraremos el problema de mezclar datos entre hilos. El problema que nos encontramos con esta solución es que abrir y cerrar cada vez las conexiones hará el programa más lento, además de ser menos eficiente, ya que establecer una conexión real con la BBDD es un proceso costoso.
- Utilizar una única conexión y sincronizar el acceso a ella desde cada uno de los hilos. Esta solución es más eficiente que la anterior, pero requiere de un esfuerzo extra por parte del programador, ya que tendremos que hacer uso del `synchronized` en cada uno de los métodos involucrados en ejecutar una sentencia sobre la BBDD. Otro problema es que los hilos tendrán que esperar para poder hacer uso de la conexión.



- Otra opción es tener varias conexiones abiertas (pool de conexiones). Al crear un pool de conexiones, tendremos disponibles varias conexiones abiertas con la BBDD. Cuando un hilo necesite una conexión, tendrá que pedirla, se le asignará y la podrá utilizar. Cuando el hilo termine, liberará dicha conexión para que la pueda usar otro hilo de la aplicación. Con esta opción, no tenemos que abrir y cerrar las conexiones cada vez que un hilo tenga que conectarse a la BBDD, tampoco tienen que esperar a que un hilo acabe con una conexión para poderla utilizar. Esta opción es la más acertada y la conocemos como pool de conexiones.

Con la tercera opción y creando la aplicación en Java, un pool de conexiones es una clase que tiene abiertas varias conexiones a la BBDD. Cuando un hilo necesite una conexión a la BBDD, en lugar de crearla con el método `DriverManager.getConnection()`, el hilo pedirá al pool la conexión a través del método `pool.getConnection()`. El pool cogerá una de las conexiones que tiene abiertas con la BBDD, la asignará al hilo, la marcará como asignada y la devolverá para que el hilo la pueda utilizar. El siguiente hilo que pida una conexión utilizando el método `pool.getConnection()`, hará que el pool busque una nueva conexión abierta disponible.



Para poder trabajar en Java con un pool de conexiones, necesitamos utilizar la librería commons.dbcp-all-1.3.jar. Esta librería la tenemos que añadir al proyecto que representa nuestra aplicación Java. Contiene las clases y métodos necesarios para poder implementar de forma sencilla un pool de conexiones.

La interfaz encargada de simular este pool de conexiones es DataSource. Esta interfaz contiene los métodos necesarios para facilitar la implementación de un pool de conexiones. Además, también necesitaremos la librería commons-pool que es la que contiene la clase BasicDataSource que es la que utilizaremos para programar ese pool. Para ello, simplemente crearemos un nuevo objeto de esta clase y pasaremos los parámetros necesarios para establecer la conexión a la BBDD. Para este paso de parámetros, utilizaremos una serie de métodos set disponibles en la propia clase.

En el siguiente ejemplo vemos el uso de un pool de conexiones:

```
public Connection crearConexion() {
    Connection conexion = null;
    BasicDataSource bdSource = new BasicDataSource();
    bdSource.setUrl("jdbc:mysql://localhost:3306/dicografica");
    bdSource.setUsername("root");
    bdSource.setPassword("");
    Connection con = null;
    try {
        if (conexion != null) {
            System.out.println("No se puede crear una nueva conexión");
        } else {
            //DataSource nos reserva una conexión y nos la devuelve
            con = bdSource.getConnection();
            System.out.println("Conexión creada satisfactoriamente");
        }
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
    }
    return con;
}
```



Recursos y Enlaces

- [Java](#)



- [API Java 11](#)



- [NetBeans](#)



- [Java JDBC API](#)





Conceptos clave

- **JDBC:** Java DataBase Connectivity. API de clases Java que permite a una aplicación Java acceder a una BBDD
- **DDL:** Data Definition Language. Lenguaje que permite crear la estructura de una BBDD.
- **DCL:** Data Control Language. Lenguaje que permite administrar el acceso a una BBDD.
- **DML:** Data Manipulation Language. Lenguaje que permite acceder y modificar los datos de una BBDD.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP



Test de autoevaluación

1. Pon el significado de cada una de las siglas de JDBC:
 - a. J
 - b. D
 - c. B
 - d. C

2. ¿Cómo se llama el lenguaje que nos permite crear sentencia para crear la estructura de una BBDD?
 - a. DLD
 - b. DCL
 - c. DDL
 - d. DML

3. ¿Qué clase representa una sentencia en la API JDBC?
 - a. Statement
 - b. Query
 - c. DriverManager
 - d. ResultSet

VERSIÓN



Ponlo en práctica

Actividad 1

Crea una aplicación en Java que:

- a. Acceda a una BBDD. Esta BBDD tiene que tener una tabla con al menos 3 columnas.
- b. La aplicación accederá a la BBDD, hará un select de todos los registros de la tabla y mostrará el resultado por pantalla.





SOLUCIONARIOS

Test de autoevaluación

1. Pon el significado de cada una de las siglas de JDBC:

- a. J – **Java**
- b. D – **Data**
- c. B – **Base**
- d. C – **Connectivity**

2. ¿Cómo se llama el lenguaje que nos permite crear sentencia para crear la estructura de una BBDD?

- a. DLD
- b. DCL
- c. **DDL**
- d. DML

3. ¿Qué clase representa una sentencia en la API JDBC?

- a. **Statement**
- b. Query
- c. DriverManager
- d. ResultSet

VER



Ponlo en práctica

Actividad 1

Crea una aplicación en Java que:

- Acceda a una BBDD. Esta BBDD tiene que tener una tabla con al menos 3 columnas.
- La aplicación accederá a la BBDD, hará un select de todos los registros de la tabla y mostrará el resultado por pantalla.

Solución:

Para crear esta aplicación Java, utilizaremos una BBDD con la siguiente estructura.

#	Nombre	Tipo	Cotejamiento	Ai
1	id	int(11)		
2	titulo	varchar(20)	latin1_swedish_ci	
3	autor	varchar(20)	latin1_swedish_ci	

El programa en Java, lo hemos estructurado de tal forma que contiene dos clases, una que tiene el método main y una segunda clase donde centralizamos los métodos propios de acceso a la BBDD, ejecución de la consulta y muestra de resultados por pantalla.

Primera clase:

```
public class AccesoJDBC {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
  
        //EN ESTE MAIN SE INVOCAN A DIFERENTES MÉTODOS SEGÚN LO QUE SE QUIERA HACER.  
        //RECUERDA:  
        // * deben estar instalados los drivers MySQL en NetBeans para que funcione  
        // * Se deben crear la estructura de tablas correspondiente que funcione.  
        // * Se debe dar acceso a MySQL con usuario "root" y clave "" para que funcione  
        Gestor_conexion gestor = new Gestor_conexion();  
  
        //USALO PARA: EJECUTAR UNA CONSULTA SELECT  
        gestor.Consulta_Statement();  
        gestor.cerrar_Conexion();  
    }  
}
```



Segunda clase:

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

public class Gestor_conexion {

    Connection conn1 = null;

    public Gestor_conexion() {
        // ABRE UNA CONEXIÓN A UNA BASE DE DATOS QUE SE SUPONE MYSQL Y QUE TIENE LAS TABLAS
        // Y LOS USUARIOS CREADOS SEGÚN ESTE EJEMPLO.
        try {
            //RECUERDA: PARA EJECUTAR ESTE CÓDIGO ES NECESARIO TENER MySQL FUNCIONANDO
            //Y LAS TABLAS Y USUARIOS CREADOS
            String url1 = "jdbc:mysql://localhost:3306/discografica";
            String user = "root";
            String password = "";
            conn1 = DriverManager.getConnection(url1, user, password);
            if (conn1 != null) {
                System.out.println("Conectado discografica");
            }
        } catch (SQLException ex) {
            System.out.println("ERROR:La dirección no es válida o el usuario y clave");
            ex.printStackTrace();
        }
    }

    public void Consulta_Statement() {
        //CRAR UN ESTAMENT PARA UNA CONSULTA SELECT
        try {
            Statement stmt = conn1.createStatement();
            String query = "SELECT * FROM album";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                System.out.println("ID - " + rs.getInt("id")
                    + ", Título " + rs.getString("titulo")
                    + ", Autor " + rs.getString("autor"));
            }
            rs.close();
            stmt.close();
        } catch (SQLException ex) {
            System.out.println("ERROR:al consultar");
            ex.printStackTrace();
        }
    }
}
```

El resultado que veremos por pantalla será el siguiente:

```
Conectado discografica
ID - 1, Título Black Ice, Autor AC/DC
ID - 2, Título Recuerdame, Autor Pablo Alboran
ID - 3, Título Black Album, Autor Metallica
ID - 4, Título Super titulo, Autor ABBA
ID - 5, Título ABC, Autor AA
ID - 6, Título Dreams, Autor The Corrs
```