UNIDAD DIDÁCTICA 2

MANEJO DE CONECTORES

MÓDULO PROFESIONAL:
ACCESO A DATOS

CESUR
Tu Centro Oficial de FP

ÍNDICE

RESUMEN INTRODUCTORIO

En esta unidad se verán los mecanismos para que una aplicación pueda comunicarse con una base de datos, desde el establecimiento de la conexión hasta la ejecución de consultas y cómo extraer los datos producidos por dichas consultas.

Se introducirán y desarrollarán los conceptos sobre conectores y, en concreto, se explicarán los diferentes tipos que tenemos con Java. A partir de estos conectores se recorrerán los diferentes mecanismos para poder interactuar contra una base de datos relacional con Java.

Por último, se estudiará cómo usar procedimientos almacenados y la realización de transacciones.

INTRODUCCIÓN

Hoy en día, prácticamente todas las aplicaciones necesitan acceder a una base de datos para almacenar o consultar información. Por ello, es fundamental conocer las posibilidades existentes a la hora de elegir una base de datos para la aplicación en función de sus características, del lenguaje de programación utilizado, etc.

Java ha ido evolucionando de forma paralela al desarrollo de la tecnología de gestión de bases de datos y, actualmente, una de las arquitecturas más maduras y usadas es la arquitectura relacional.

Con Java tenemos la ventaja de trocear nuestra actividad contra la base de datos, lo que nos proporciona mucha flexibilidad. Conexión, lectura, inserción, modificación y otras sentencias son las que necesitamos habitualmente cuando realizamos desarrollo de software que impliquen almacenar información de una forma persistente en una base de datos.

CASO INTRODUCTORIO

Trabajas como desarrollador junior en una mediana empresa dedicada a proyecto de construcción y obra pública. Dentro de la empresa tienen contratado diversos software y herramientas empresariales, sin embargo, tu departamento se encarga de realizar desarrollos a medida pequeños para problemáticas muy particulares.

En particular, se quiere comenzar a mantener una base de datos de las matrículas y características de los diferentes vehículos y sus fechas de ITV. Por ello, te planteas qué arquitectura emplear, el tipo de base de datos y los conectores óptimos para Java, ya que las aplicaciones realizadas dentro de la empresa son Java.

Al finalizar la unidad, serás capaz de identificar los tipos de bases y conectores óptimos, entre otros aspectos.

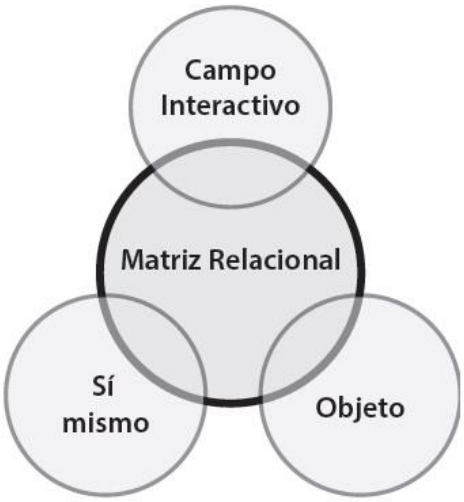
1. GESTIÓN DE LOS CONECTORES

El primer paso para comenzar a realizar la aplicación es conocer el tipo de sistema de gestión de base de datos y el conector adecuado para ese sistema de gestión de base de datos.

Al estar el sistema alojado en un taller, sabes que puedes almacenarlo en una base de datos en sus servidores. Para ello vas a utilizar una base de datos relacional (MySQL, PostgreSQL). Tienes en cuenta que, actualmente, la mejor forma de conectarse a SGBD relacionales con Java es a través del JDBC.

A pesar de que, durante los primeros años de la historia de la informática, la persistencia de los datos ha ido pasando, de década en década, por diferentes paradigmas de representación y almacenamiento de datos, pero esa tendencia al cambio se ha ido desvaneciendo con el crecimiento del **paradigma relacional.**

Se trata de una tecnología sencilla pero muy eficiente, que ha sabido adaptarse a la mayoría de los sistemas de datos que las empresas reclamaban y a un coste bastante asequible como para tomar la hegemonía absoluta del mercado desde el último cuarto del pasado siglo.



Dimensiones de la matriz relacional

Fuente: http://pepsic.bvsalud.org/scielo.php?script=sci_arttext&pid=S2145-48922014000100009&lng=pt&nrm=iso

ARTÍCULO DE INTERÉS

Stephen Mitchell y el paradigma relacional en psicoanálisis:

Es cierto que el modelo relacional tiene limitaciones importantes en la hora de representar información poco estructurada, o estructuras excesivamente dinámicas y complejas. La principal razón en su consistencia la encontramos en la solidez y madurez que los sistemas gestores de bases de datos relacionales tienen.

De hecho, hoy en día los modelos no relacionales han hecho, a su vez, evolucionar la programación y los sistemas de gestión de bases de datos, pero sin eliminar totalmente el paradigma relacional, perfectamente utilizable para problemas más sencillos.

1.1 Gestores de bases de datos embebidos e independientes

Un sistema gestor de base de datos (**SGBD**) es un conjunto de programas cuya función es administrar y gestionar la información contenida una base de datos.

Además, los sistemas de gestión de bases de datos proporcionan diferentes niveles de abstracción de la información, dependiendo del tipo de usuario que la gestiona. Normalmente, el usuario no ve detalles sobre la forma y el lugar en que están almacenados los datos, así como los procedimientos para recuperar y actualizar la información.

ENLACE DE INTERÉS

Las bases de datos NoSQL puede que sean la gran alternativa actual a las bases de datos relacionales. Accede para encontrar aquella que se adecua a tus requisitos:

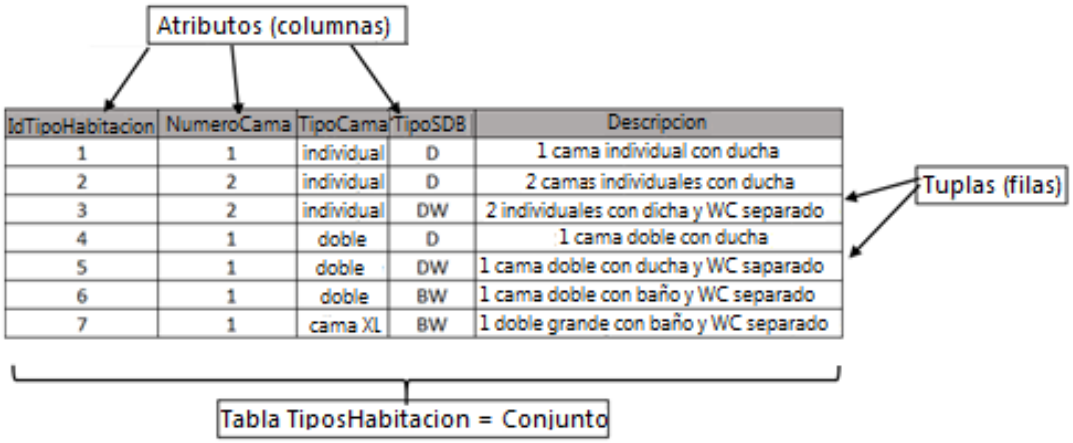
Se pueden distinguir dos tipos de sistemas de gestión de bases de datos en función de si las bases de datos son independientes o no.

- Un sistema de gestión de bases de datos **embebido** es aquel que no está implementado de manera independiente al programa con el que se establece la comunicación, sino que forma parte de él, integrándose en su estructura. Un ejemplo de SGBD embebido sería SQLite, el cual permite la gestión de bases de datos relacionales.

- Por el contrario, un SGBD **independiente** es aquel que se ejecuta de manera separada de la aplicación con la que se comunica, es decir, el sistema operativo ejecuta ambos como procesos distintos. Los SGBD independientes presentan la ventaja de que se pueden configurar y administrar de manera independiente.

1.1.1 Modelo relacional

El modelo relacional es una forma de organizar los datos de una aplicación agrupándolas según la relación que se pueda establecer entre ellas de acuerdo con el modelo. Esta agrupación se materializa en forma de tabla, donde distinguimos las columnas o conjunto de datos que representan un mismo concepto del modelo de datos, y las tuplas o registros que representan entidades diferenciadas a la aplicación.



IdTipoHabitacion	NumeroCama	TipoCama	TipoSDB	Descripcion
1	1	individual	D	1 cama individual con ducha
2	2	individual	D	2 camas individuales con ducha
3	2	individual	DW	2 individuales con dicha y WC separado
4	1	doble	D	1 cama doble con ducha
5	1	doble	DW	1 cama doble con ducha y WC saparado
6	1	doble	BW	1 cama doble con baño y WC separado
7	1	cama XL	BW	1 doble grande con baño y WC separado

Modelo relacional

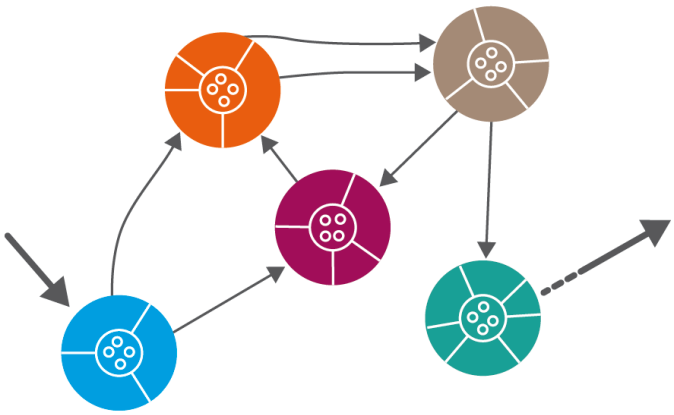
Fuente: https://www.ediciones-eni.com/open/mediabook.aspx?idR=aad70223517088b8f114ad6313723e83

Para la representación de información más compleja necesitaremos múltiples tablas que se relacionaran entre sí mediante claves foráneas.

1.1.2 Modelo orientado al objeto

Los objetos pueden representar cualquier elemento del modelo conceptual, una entidad, una característica, un proceso, una acción, una relación… Los objetos son fruto de un proceso de abstracción centrado en las características importantes (datos), pero también en el comportamiento o la funcionalidad que tendrán en el momento de materializarse durante la ejecución de las aplicaciones (código).

La importancia de centrar el modelo en los objetos es múltiple. En referencia a los datos, los objetos actúan como estructuras jerárquicas, de forma que la información queda siempre perfectamente contextualizada dentro de los objetos.



Modelo orientado a objetos

Fuente: https://ioc.xtec.cat/materials/FP/Materials/2252_DAM/DAM_2252_M06/web/html/WebContent/u2/a1/continguts.html

En el paradigma OO, la interacción entre objetos queda totalmente pautada a través del tipo de relaciones que establecemos en el modelo. Hablaremos de un objeto

relacionado con otro cuando lo primero pueda acceder a los métodos del segundo. Las relaciones definidas al modelo OO indicarán qué tipos de objetos podrán interactuar y cómo.

EJEMPLO PRÁCTICO

En la empresa donde trabajamos queremos desarrollar una aplicación para la gestión de vehículos. Las especificaciones que nos trasladan son:

- Se crearán del entorno de 10 a 15 tablas para la gestión de la aplicación.
- El número de usuarios no superará los 20 usuarios simultáneos.
- Necesitamos poder incorporar roles.
- El gestor de base de datos debe ser gratuito.

¿Qué modelo y sistema de gestión de base de datos elegir?

A partir de las especificaciones podemos descartar el modelo no relacional, ya que la cantidad de información, usuario y la evolución futura va a ser pequeña.

Respecto a las bases de datos relacionales, podemos reducir nuestro espectro a dos posibilidades, que cumplen las especificaciones y, además, son sistemas muy maduros:

- PostgreSQL.
- MariaDB.

Cualquiera de las dos opciones será totalmente válido para nuestras especificaciones.

1.2 El desfase objeto-relacional

El desfase objeto-relacional hace referencia a las diferencias existentes entre la programación orientada a objetos (POO) y las bases de datos relacionales. En ocasiones, nos lo podemos encontrar llamado impedancia objeto-relacional.

Estas diferencias se refieren a aspectos como:

- Los tipos de datos difieren, ya que en la POO se pueden utilizar tipos complejos y, sin embargo, en las bases de datos se tienen que limitar a tipos de datos básicos.

- En la POO se trabaja con el concepto denominado **objeto**. Sin embargo, en las bases de datos relacionales se manejan **tablas y tuplas**, por lo que hay que

realizar una traducción del modelo orientado a objetos al modelo entidad relación.

- Además, en la POO se habla de **asociaciones** entre objetos. Sin embargo, el modelo relacional trabaja con las **relaciones** entre tablas.

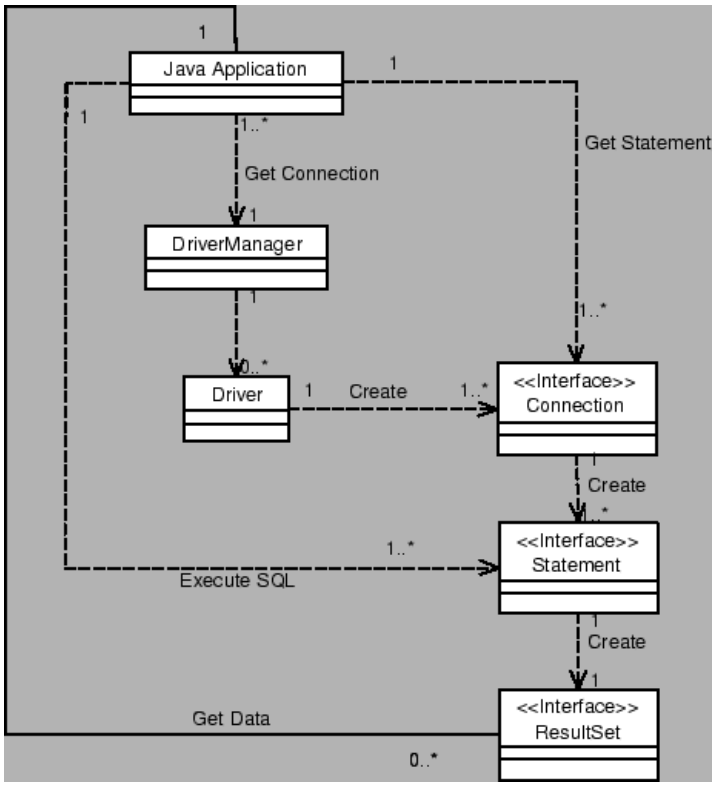
Aunque se podrían citar muchas más, en términos generales se puede decir que la impedancia objeto-relacional se refiere al conjunto de dificultades técnicas que se encuentran cuando se escriben programas orientados a objetos que acceden a bases de datos relacionales. Esto ocurre a menudo, ya que las bases de datos relacionales son las más comunes, y el paradigma de la POO es uno de los más frecuentemente usados en la programación moderna.

1.3 Protocolos de acceso a bases de datos. Conectores. Establecimiento de conexiones

La información almacenada en las bases de datos es casi siempre el bien más preciado para una empresa. Normalmente, cada proveedor de SGBD (Oracle, Sybase, Informix, etc.) ofrece su propio protocolo para acceder a sus bases de datos. Las aplicaciones cliente escritas en lenguajes nativos pueden utilizar estos protocolos para acceder a los datos, pero esto presenta la desventaja de que no se ofrece una interfaz común para el acceso a diferentes bases de datos, por lo que habría que cambiar la implementación de la aplicación cliente en caso de cambiar de base de datos.

Existen alternativas al uso de estos protocolos, que permiten acceder a distintas bases de datos tan solo cambiando el controlador que se encarga de comunicarse con la base de datos. Por ejemplo, en el caso de la plataforma Java, bastará con hacer uso del driver JDBC que ofrezca el proveedor del servicio al que se quiere acceder.

JDBC (Java DataBase Connectivity) es la **API** que permite la conexión con las bases de datos utilizando el lenguaje de programación Java, independientemente del sistema operativo que se utilice y de la base de datos a que se quiera acceder. Para esto, es necesario que el proveedor ofrezca un driver JDBC que cumpla con dicha especificación y traduzca las llamadas JDBC a las APIs correspondientes.



Arquitectura driver JDBC

Fuente: Coremsa

PARA SABER MÁS

En la web de Oracle se puede encontrar la documentación oficial sobre la API JDBC:

Se pueden distinguir cuatro tipos de drivers JDBC:

- **Tipo 1:** driver puente JDBC-ODBC.
- **Tipo 2:** driver API nativo / parte Java.
- **Tipo 3:** driver protocolo de red / todo Java.
- **Tipo 4:** driver protocolo nativo / todo Java.

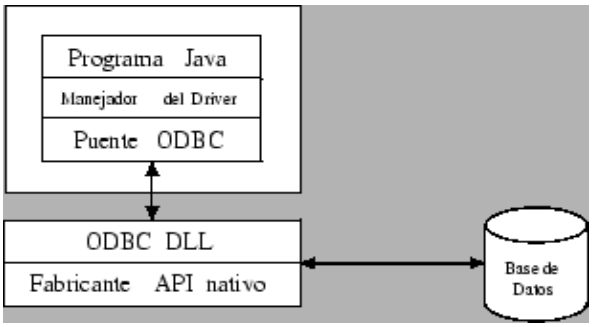
VÍDEO DE INTERÉS

Atiende a este vídeo sobre el uso de JDBC con Java:

1.3.1 Tipo 1: driver puente JDBC-ODBC

Este tipo de drivers se encargan de traducir operaciones JDBC en llamadas a la API de Microsoft ODBC.

Las llamadas a la API ODBC son ejecutadas sobre la base de datos utilizando el driver ODBC que corresponda.



Arquitectura driver JDBC-ODBC
Fuente: Coremsa

ENLACE DE INTERÉS

Obtén más información sobre el estándar ODBC:

PARA SABER MÁS

Conoce más información sobre este puente, JDBC-ODBC:

Este tipo de driver presenta algunas **ventajas**:

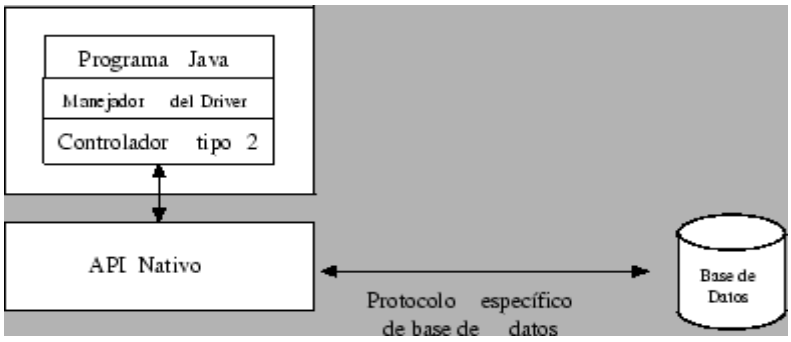
- Permite trabajar con multitud de drivers ODBC.
- Son útiles para las empresas que disponen de un driver ODBC instalado, ya que cuando se programa con un lenguaje de programación distinto a Java normalmente se utiliza ODBC para realizar la conexión con las bases de datos.
- En ocasiones, es la única forma de conectar con determinadas bases de datos como, por ejemplo, Microsoft Access.

Sin embargo, existen también ciertas **desventajas** que hacen que no siempre sea el driver más apropiado:

- Debido al número de traducciones que hay que realizar, este driver es el que menos rendimiento proporciona.
- Este tipo de drivers no funcionan adecuadamente con applets.
- La mayoría de los navegadores no tienen soporte nativo para este puente.

1.3.2 Tipo 2: driver API nativo / parte Java

Este tipo de drivers se encargan de traducir las solicitudes de API JDBC en llamadas específicas a bases de datos para RDBMS como, por ejemplo, SQL Server, utilizando la interfaz de métodos nativos que proporciona Java.



Arquitectura driver JDBC nativo

Fuente: Coremsa

Como principal ventaja de este tipo de drivers hay que destacar su rendimiento. Sin embargo, presenta las mismas desventajas que el tipo 1, al hacer que la interfaz de conectividad deba estar previamente instalada en la máquina cliente.

Aunque los drivers de tipo 2 habitualmente ofrecen mejor rendimiento que el puente JDBC-ODBC, ya que las llamadas JDBC no tienen que convertirse en llamadas ODBC, estos ofrecen menos rendimiento que los tipos siguientes. Además, siguen teniendo los mismos problemas que el tipo anterior, puesto que la interfaz de conectividad nativa debe estar previamente instalada en la máquina cliente.

Una última característica que se debe destacar es que los drivers tipo 1 suelen estar escritos en alguna combinación de Java y C/C++, ya que el driver debe utilizar una capa de C para realizar llamadas a la biblioteca, que está escrita en este mismo lenguaje.

En resumen, este tipo presenta algunas **ventajas**:

- **Mejor rendimiento que el driver tipo 1:** no requiere la conversión de las llamadas JDBC a ODBC, lo que mejora la eficiencia.

- **Compatibilidad con RDBMS específicos:** estos drivers están optimizados para funcionar con sistemas de gestión de bases de datos relacionales específicos, lo que permite un rendimiento más rápido y eficiente.

- **Uso de la interfaz de métodos nativos:** aprovecha las capacidades nativas del sistema operativo, lo que puede ofrecer mejoras de velocidad y optimización.

Sus **desventajas** son:

- **Dependencia de la instalación en el cliente:** la interfaz de conectividad nativa debe estar instalada en la máquina cliente, lo que puede complicar la configuración y la portabilidad.

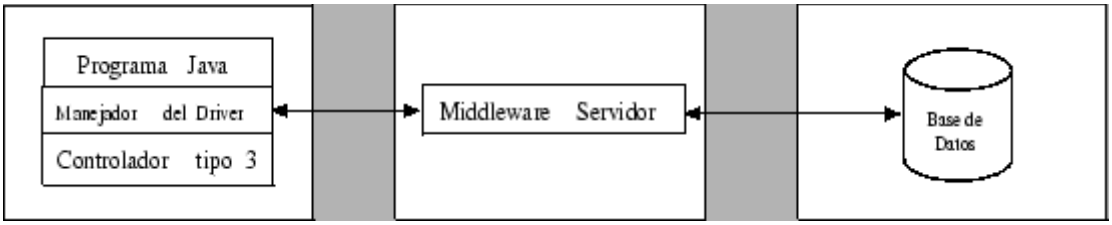
- **No es completamente multiplataforma:** debido al uso de componentes nativos, la compatibilidad entre diferentes sistemas operativos puede verse limitada.

- **Interfaz nativa dependiente del fabricante:** se requiere que el proveedor de la base de datos ofrezca un driver compatible, lo que puede ser un inconveniente si se utilizan múltiples bases de datos.

1.3.3 Tipo 3: driver protocolo de red / todo Java

Este tipo de drivers están implementados utilizando una **arquitectura de tres capas**:

1. Capa cliente JDBC.
2. Capa intermedia.
3. Base de datos a la que se accede.



Arquitectura driver JDBC red

Fuente: Coremsa

Esta arquitectura permite que las solicitudes a la base de datos sean traducidas en un protocolo de red independiente de la base de datos que se vaya a utilizar. En este modelo de tres capas existe una capa intermedia que se encarga de recibir las solicitudes y enviarlas a la base de datos, utilizando para esto un driver tipo 1 o 2.

Este tipo de driver presenta algunas **ventajas:**

- **Independencia de la plataforma:** al estar escrito completamente en Java, estos drivers son independientes de la plataforma, lo que facilita su uso en diferentes sistemas operativos.

- **No requiere bibliotecas nativas en el cliente:** la presencia de una capa intermedia elimina la necesidad de instalar bibliotecas específicas del fabricante en la máquina cliente, lo que simplifica la configuración.

- **Mejor rendimiento en entornos distribuidos:** La arquitectura de tres capas permite optimizar el rendimiento en aplicaciones donde la base de datos no se encuentra localmente.

Sin embargo, existen también ciertas **desventajas** que hacen que no siempre sea el driver más apropiado:

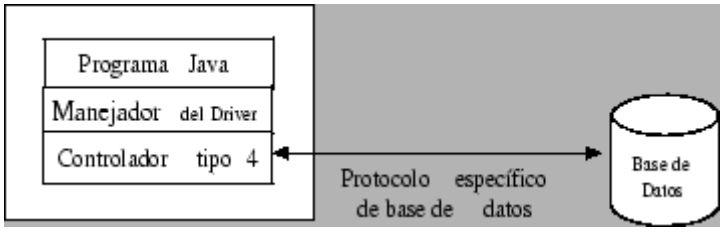
- **Dependencia del código específico en la capa intermedia:** la capa intermedia debe ser capaz de manejar el protocolo de red y las comunicaciones con la base de datos, lo que puede complicar el desarrollo y la gestión del sistema.

- **Complejidad de implementación:** implementar y mantener una capa intermedia puede ser más complicado y requerir un mayor esfuerzo de desarrollo y administración.

1.3.4 Tipo 4: driver protocolo nativo / todo Java

Este tipo de driver realiza la conexión con las bases de datos de manera directa utilizando, para esto, el protocolo nativo del servidor. Por tanto, no necesitan utilizar ODBC o APIs nativas.

Como principales características, cabe destacar que están escritos 100% en lenguaje Java, por lo que son independientes de la plataforma y eliminan todos los aspectos de configuración en el cliente.



Arquitectura driver JDBC totalmente nativo Java

Fuente: Coremsa

Sus **ventajas** son:

- **Rendimiento superior:** ofrecen el mejor rendimiento de todos los tipos de drivers JDBC, al conectar directamente con la base de datos utilizando el protocolo nativo del servidor.

- **Total independencia de la plataforma:** al estar escritos completamente en Java, son totalmente independientes del sistema operativo y no requieren instalaciones adicionales en el cliente.

- **Fácil configuración:** no requieren configuración adicional en el lado del cliente, lo que simplifica su uso y despliegue en entornos variados.

- **Compatibilidad con aplicaciones Java puras:** dado que están escritos en Java, son ideales para aplicaciones Java que necesitan conectarse a bases de datos sin complicaciones adicionales.

Sus **desventajas:**

- **Específicos del fabricante de bases de datos:** estos drivers están diseñados para funcionar con un único sistema de gestión de bases de datos, lo que puede limitar su uso en entornos donde se manejan múltiples tipos de bases de datos.

- **No son universales:** la necesidad de un driver diferente para cada tipo de base de datos puede ser un inconveniente en sistemas que requieren flexibilidad y compatibilidad con múltiples bases de datos.

PARA SABER MÁS

Aquí tienes un tutorial sobre cómo se pueden crear drivers propios JDBC:

ENLACE DE INTERÉS

Conoce más información sobre los cuatro tipos de drivers JDBC existentes:

2. SENTENCIAS Y TRANSACCIONES CON LAS BASES DE DATOS

Una vez que has elegido MariaDB (MySQL) como tu gestor de bases de datos, y teniendo en cuenta que en la empresa la aplicación debe ser desarrollada con el lenguaje Java, tienes que comenzar a planificar como gestionar las diferentes estrategias de interacción con la base de datos.

Debes establecer los mecanismos de conexión, de lectura, de escritura y de modificación de datos en la base de datos. Este punto de partida te permitirá finalizar el inicio del proyecto planteado para la gestión de vehículos.

Ahora veremos los elementos básicos del API JDBC que permiten a las aplicaciones Java comunicarse con un SGBD usando el lenguaje SQL. Hace falta, por lo tanto, que dispongamos del conector JDBC de PostgreSQL o de MariaDB (MySQL), dependiendo de la base de datos elegida y que lo añadamos en las bibliotecas de nuestro proyecto. También será necesario que habilitemos una conexión para consultar la base de datos sin salir de la IDE.

Es importante crear un proyecto de prueba, junto a una base de datos paralela para realizar las depuraciones oportunas antes de comenzar a interactuar con la base de datos definitiva. Este proceso también se puede realizar en el propio proyecto contra una base de datos de prueba y creando ficheros de configuración que nos permitan cambiar de una base de datos de prueba a la de producción.

El proceso de trabajo, con una base de datos y la interacción a través de un conector, es un proceso que podemos extrapolar a otros proyectos, y que incluso podríamos crear clases y paquetes en Java que, dentro de nuestro departamento o proyectos, fueran reutilizados.

2.1 Establecimiento de conexiones

El API JDBC permite realizar las conexiones de **dos formas distintas:**

- Utilizando la clase java.sql.DriverManager, la cual permite acceder a bases de datos desde aplicaciones escritas en Java.

- Desde aplicaciones J2EE (Java 2 Enterprise Edition).

En esta unidad se estudiará la primera de las opciones. Lo primero que hay que indicar es que desde una aplicación es posible realizar una o más conexiones con una o varias bases de datos utilizando los drivers JDBC. En este caso, cada driver implementa la interfaz java.sql.Driver. Para realizar la conexión con la base de datos se hará uso del método connect(), que permite establecer una conexión con la base de datos y obtener un objeto del tipo Connection.

La clase java.sql.DriverManager puede gestionar uno o varios drivers.

PARA SABER MÁS

Aquí conocerás más a fondo sobre la interfaz Connection con sus métodos y atributos:

Antes de profundizar en cómo llevar a cabo las conexiones, es necesario ver el concepto de URL en JDBC. Una URL de JDBC permite identificar los drivers de una base de datos, así como la información adicional necesaria para localizar la base de datos y realizar la conexión con ella. La sintaxis de las URLs es la siguiente:

```
jdbc:<subprotocolo>:<subnombre>
```

Existen **tres partes** separadas por dos puntos y, de las cuales, explicamos que indica cada una:

1. **jdbc:** indica el protocolo.

2. **Subprotocolo:** indica el driver utilizado para acceder a la base de datos.

3. **Subnombre:** indica información específica para localizar y acceder a la base de datos. Esta parte puede incluir detalles como el host, puerto y nombre de la base de datos, y puede variar dependiendo del subprotocolo.

Por ejemplo, para una base de datos MySQL llamada ‘empleados’ en el propio ordenador, la URL sería la siguiente:

```
jdbc:mysql://localhost:3306/empleados
```

Y si, por ejemplo, se estuviera utilizando Oracle, mediante el puente JDBC-ODBC, la URL sería:

```
jdbc:odbc:empleados
```

Volviendo a la clase DriverManager, su propósito es suministrar una capa común encima de los distintos drivers de bases de datos utilizados en una aplicación. Además, esta clase necesita que cada driver de los que haga uso la aplicación sea registrado antes de utilizarlo, para que DriverManager lo reconozca.

La forma de **registrar un driver** es la siguiente:

```
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
    // Driver no encontrado
```

Como hemos comentado, una vez registrados los drivers ya es posible utilizar los métodos disponibles para la conexión con la base de datos.

ENLACE DE INTERÉS

Aquí encontrarás la documentación oficial sobre la clase DriverManager:

PARA SABER MÁS

También es interesante que veas un ejemplo de conexión a MySQL, usando las clases Class, Connection, DriverManager, ClassNotFoundException y SOLException:

El método utilizado para realizar la conexión es getConnection() cuya sintaxis es la siguiente:

```
public static Connection getConnection(String url, java.util.Properties info) throws SQLException
```

Donde:

- **url:** indica la url de la base de datos según lo estudiado anteriormente (jdbc:subprotocolo:subnombre).

- **Properties:** objeto que contiene los parámetros de autenticación requeridos por el servidor de la base de datos. Las propiedades variarán en función de la base de datos.

Como resultado de ejecutar este método se obtiene el driver apropiado del conjunto de drivers registrados.

EJEMPLO PRÁCTICO

uestro jefe, en el departamento de desarrollo de la empresa donde trabajamos, nos ha indicado que, antes de comenzar a desarrollar la interacción con la base de datos, realicemos una clase y unos métodos con Java que nos permitan configurar el establecimiento de conexión con Java y que, a su vez, sea suficientemente modular para poderlo usar con otros proyectos. ¿Cómo nos plantearíamos esta clase y los métodos?

1. El primer paso es crear una nueva clase denominada DB dentro de un paquete llamado Model y con una nueva clase.

```
package Model;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Date;

public class DB {
```

2. En segundo lugar, crearemos la zona de propiedades necesarias para la conexión y uso de la base de datos.

```
  private Connection connect = null;
  private Statement statement = null;
  private PreparedStatement preparedStatement = null;
 private ResultSet resultSet = null;
```

3. En tercer lugar, crearemos el método que nos permite conectarnos.

```
    public void conexionDB() throws Exception {
        try {
            // Carga el driver
            Class.forName("com.mysql.jdbc.Driver");
            // Çonfigura la conexión
            connect = DriverManager

.getConnection("jdbc:mysql://localhost/feedback?"
                            + "user=sqluser&password=sqluserpw");
        } catch (Exception e) {
            throw e;
        } finally {
            close();
        }
```

4. En último lugar crearemos el método que nos permite cerrar la conexión.

```
    private void close() {
        try {
            if (resultSet != null) {
                resultSet.close();
            }

            if (statement != null) {
                statement.close();
            }
```

```
            if (connect != null) {
                connect.close();
            }
        } catch (Exception e) {

        }
    }
```

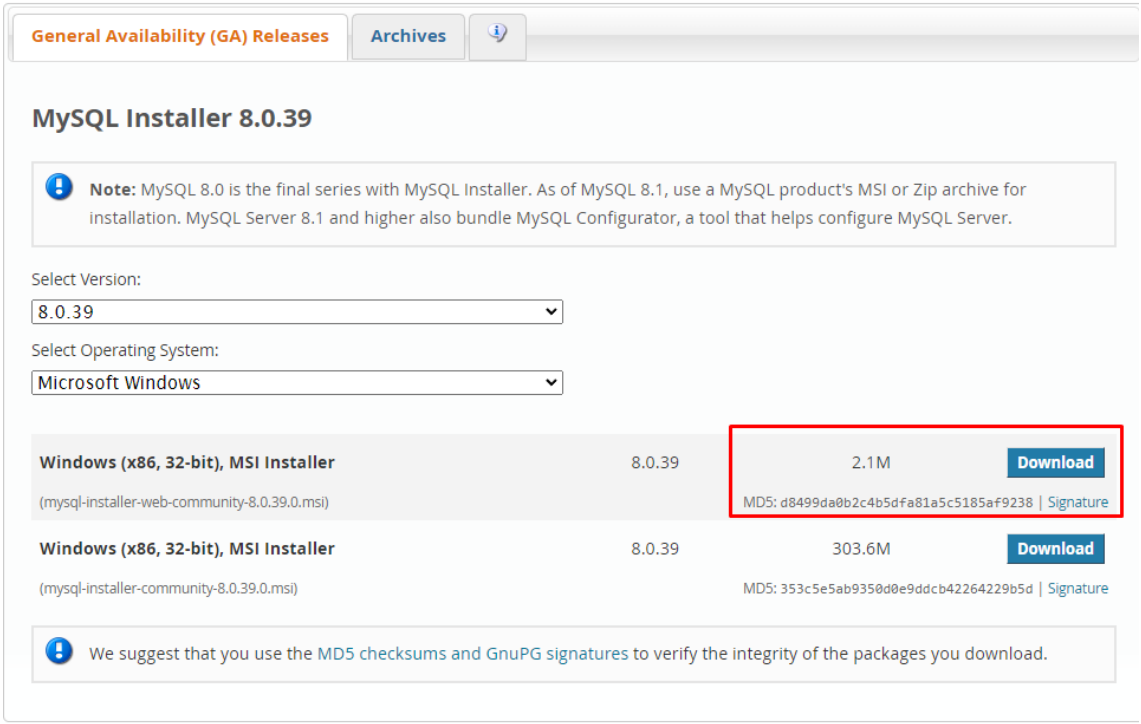
VÍDEO DE INTERÉS

Visualiza un ejemplo sobre el establecimiento de la conexión.

2.2 Agregando JDBC de MySQL a nuestro proyecto Java

Como hemos visto en apartado anteriores, tenemos multitud de drivers JDBC para acceder a nuestras bases de datos. En este apartado vamos a ver cómo agregar el JDBC de MySQL.

Para agregar el JDBC a nuestro proyecto Java debemos acceder a la página oficial de MySQL y seleccionar la versión más moderna (8.0.39) y nuestro sistema operativo (en este caso Windows), a continuación, podemos elegir cualquiera de los dos instaladores, aunque se recomienda el de menor peso.

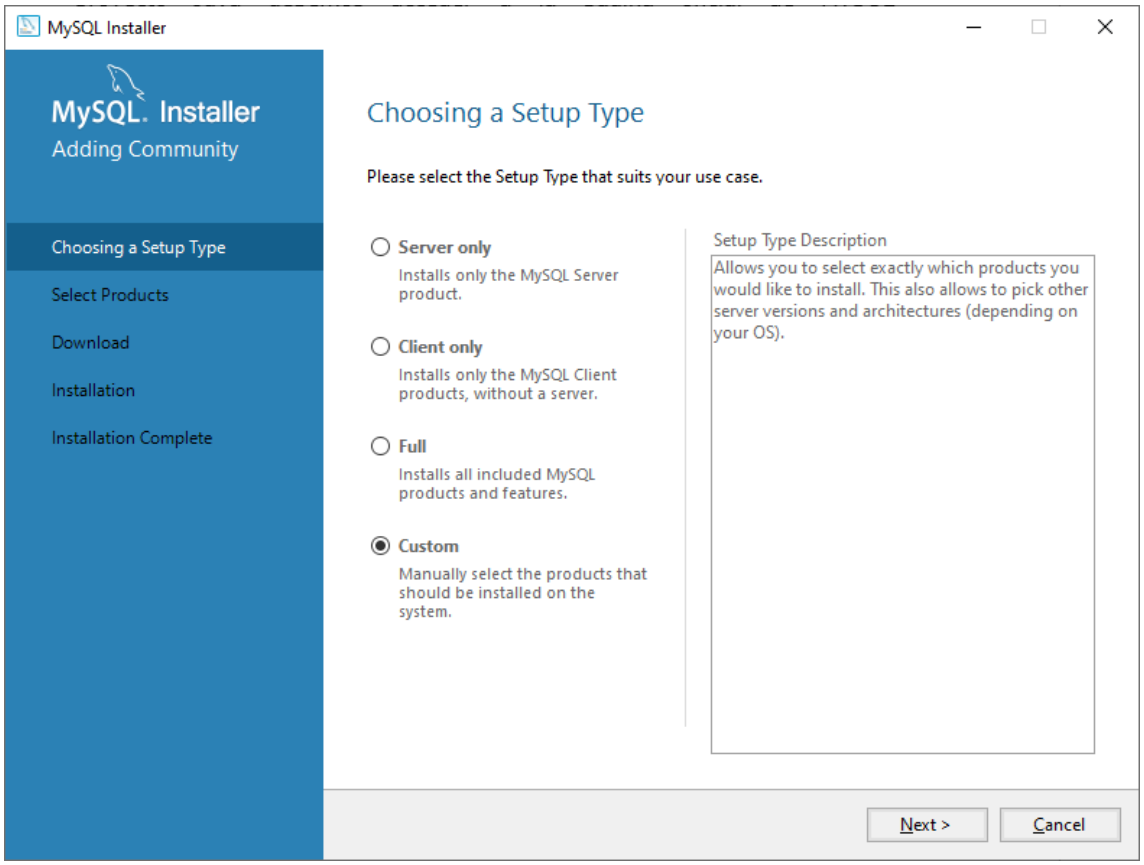


Driver JDBC MySQL
Fuente: Elaboración propia

ENLACE DE INTERÉS

Aquí podrás seleccionar y descargar dicha versión de MySQL:

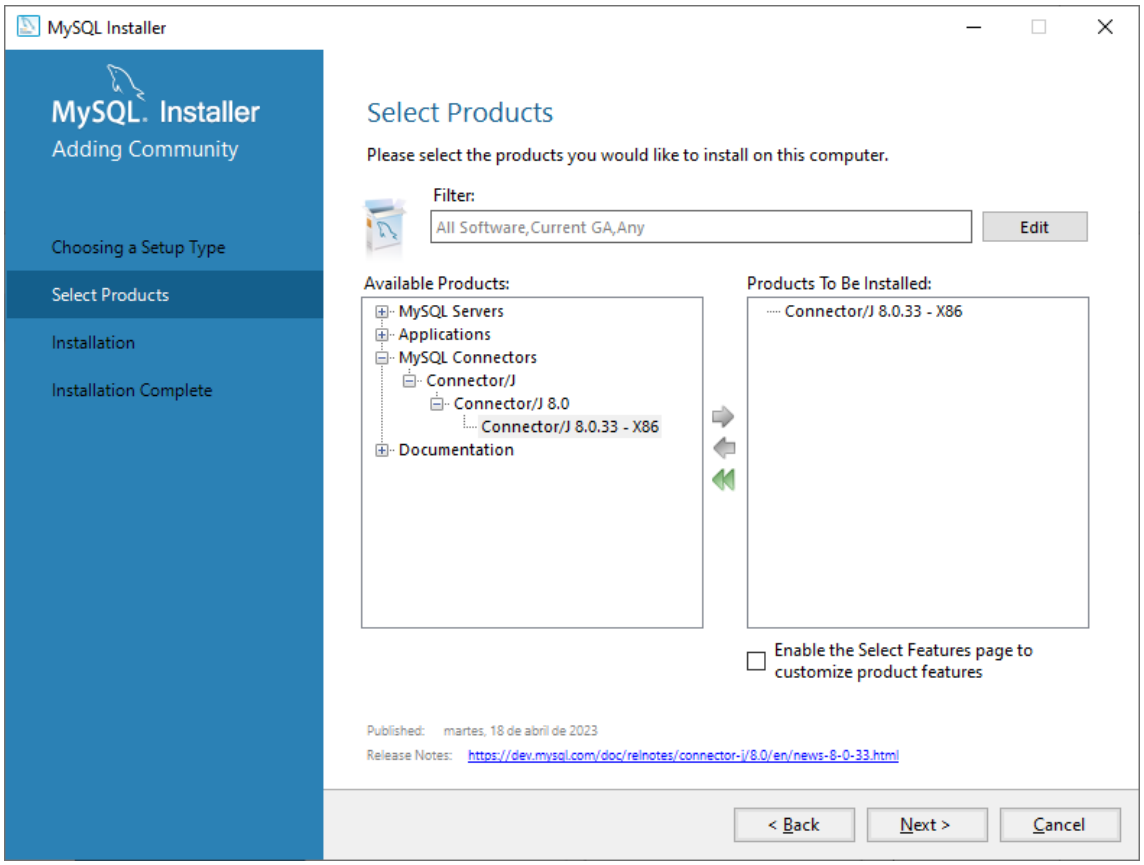
Una vez abramos el instalador marcamos “Custom”, ya que solo vamos a instalar el JDBC y no todo el software.



Instalación JDBC Paso 1

Fuente: Elaboración propia

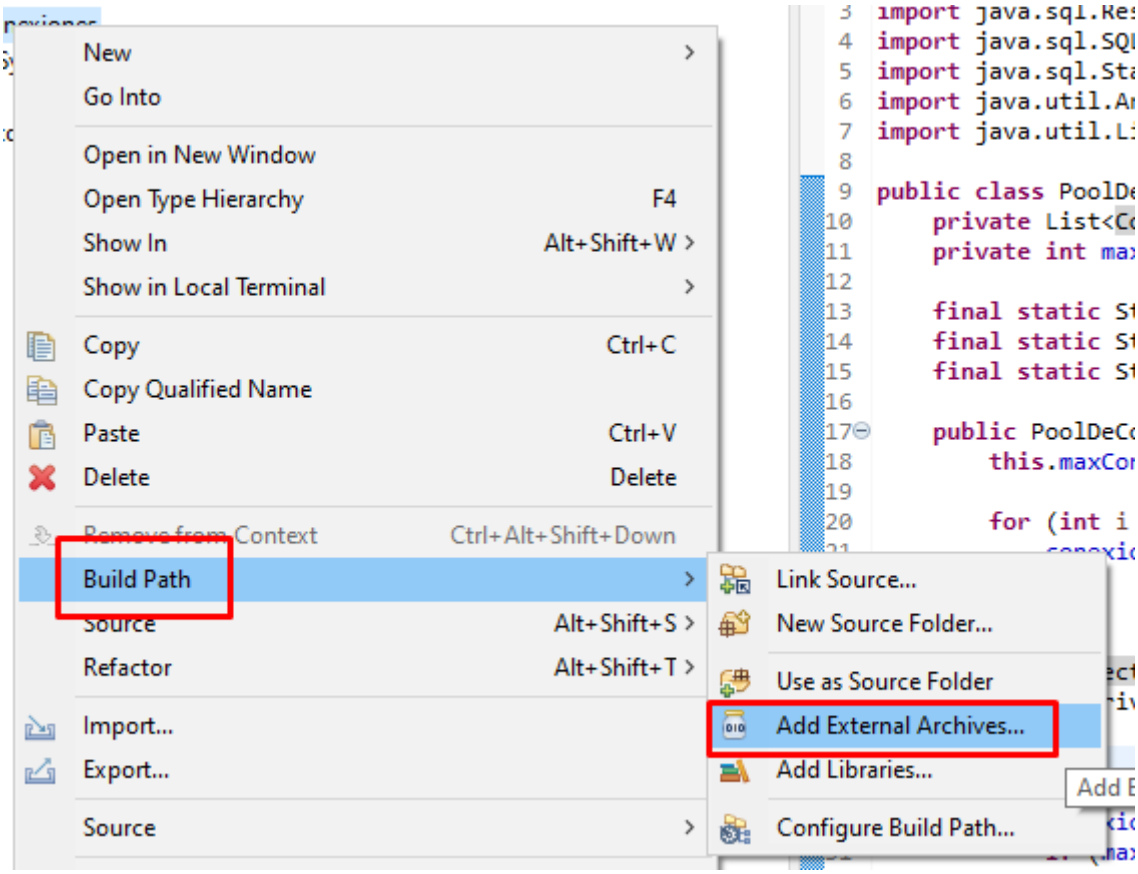
Luego “MySQL Connectors” y el Conector de Java.



Instalación JDBC Paso 2

Fuente: Elaboración propia

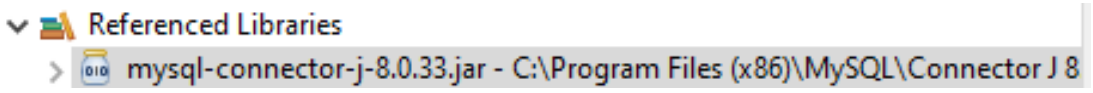
Una vez lo instalamos encontraremos el fichero JAR en la ruta C:\Program Files (x86)\MySQL\Connector J 8.0. Para agregarlo a un proyecto de Eclipse pulsamos botón derecho en el proyecto y “Build Path -> Add External Archives…”, ahí seleccionaremos el jar del JDBC.



Agregando JDBC a nuestro Proyecto

Fuente: Elaboración propia

Por último, podremos ver nuestro conector en el apartado “Referenced Libraries” de nuestro proyecto.



JDBC en nuestro proyecto Java

Fuente: Elaboración propia

2.3 Pool de Conexiones

Muchas veces sucede que abrimos y cerramos conexiones, lo que resulta ineficiente para el sistema que accede a los datos. El pooling de conexiones es una técnica utilizada para mejorar el rendimiento y la eficiencia en el manejo de conexiones a bases de datos. En lugar de abrir y cerrar una nueva conexión para cada solicitud, se reutilizan conexiones existentes.

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

```
import java.util.ArrayList;
import java.util.List;

public class PoolDeConexionesSimple {
    private List<Connection> conexionesDisponibles = new
ArrayList<>();
    private int maxConexiones;

    final static String url = "jdbc:mysql://localhost:3306/clase";
    final static String usuario = "root";
    final static String contraseña = "";

    public PoolDeConexionesSimple(String url, String usuario, String
contraseña, int conexionesIniciales, int maxConexiones) throws
SQLException {
        this.maxConexiones = maxConexiones;

        for (int i = 0; i < conexionesIniciales; i++) {
            conexionesDisponibles.add(crearNuevaConexion(url, usuario,
contraseña));
        }
    }

    private Connection crearNuevaConexion(String url, String usuario,
String contraseña) throws SQLException {
        return DriverManager.getConnection(url, usuario, contraseña);
    }

    public synchronized Connection obtenerConexion() throws
SQLException {
        if (conexionesDisponibles.isEmpty()) {
            if (maxConexiones > 0 && conexionesDisponibles.size() <
maxConexiones) {
                return crearNuevaConexion(url, usuario, contraseña);
            } else {
                throw new SQLException("Se alcanzó el número máximo de
conexiones");
            }
        } else {
            return
conexionesDisponibles.remove(conexionesDisponibles.size() - 1);
        }
    }

    public synchronized void liberarConexion(Connection conexion) {
        conexionesDisponibles.add(conexion);
    }

    public static void main(String[] args) {
        try {
            PoolDeConexionesSimple pool = new
PoolDeConexionesSimple(url, usuario, contraseña, 5, 20);
```

```
            Connection conn = pool.obtenerConexion();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM
empleados");

            while (rs.next()) {
                int id = rs.getInt("id");
                String nombre = rs.getString("nombre");
                System.out.println("ID: " + id + ", Nombre: " +
nombre);
            }

            pool.liberarConexion(conn);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```


ENLACE DE INTERÉS

Conoce más sobre la técnica de pooling para las conexiones a la base de datos:

2.4 Ejecución de sentencias de definición de datos

Una vez que se ha establecido la conexión con la base de datos ya es posible trabajar con ella para insertar, actualizar o eliminar datos.

Para esto, se debe hacer uso de la clase Statement de Java, cuyos métodos más importantes son los siguientes:

- **ExecuteUpdate():** permite la ejecución de sentencias para la modificación de la base de datos como, por ejemplo: INSERT, UPDATE, DELETE, etc.
- **ExecuteQuery():** permite la ejecución de consultas sobre la base de datos.

A pesar de que SQL es, en esencia, un lenguaje de consulta, también incluye unas cuántas órdenes imperativas que permiten hacer peticiones para cambiar las estructuras internas del SGBD donde se almacenarán los datos (instrucciones conocidas con las siglas DDL, de la inglés Data Definition Language), otorgar permisos a los usuarios existentes o crear de nuevos (subgrupo de instrucciones conocidas como DCL o Data Control Language) o incluso modificar los datos almacenados usando las instrucciones insert, update y delete.

Aunque se tratan de sentencias muy dispares, desde el punto de vista de la comunicación con el SGBD, se comportan de manera muy parecida, siguiendo el patrón siguiente:

- Instanciación a partir de una conexión activa.
- Ejecución de una sentencia SQL pasada por parámetro al método executeUpdate/executeQuery.
- Cierre del objeto Statement instanciado.

Vemos un ejemplo de obtención y ejecución de un Statement a partir de un objeto conexión referenciado por la variable *con*:

```
...
 Statement statement = con.createStatement();
statement.executeUpdate(sentenciaSQL);
statement.close();
```

Cuando hablamos de definición de datos, en términos de Bases de datos, tenemos que hablar de las siguientes sentencias SQL principalmente:

- **CREATE:** sirve para crear una nueva base de datos, tabla, índice y otros. En concreto, CREATE TABLE nos permite crear una nueva tabla.

```
CREATE TABLE[nombre de la tabla] ( [definiciones de columna] )
[parámetros de la tabla]
```

- **DROP:** sirve para borrar objetos de la base de datos.

```
DROP objeto_a_eliminar;
DROP TABLE     myTable;
DROP SEQUENCE mySequence;
DROP INDEX     myIndex;
```

- **ALTER:** sirve para modificar objetos de la base de datos. ALTER TABLE, por ejemplo, puede modificar e interactuar con objetos de una tabla.

A continuación, podemos ver un ejemplo con Java de creación de tabla y por lo tanto de uso de estas instrucciones SQL:

```
…
private static final String CREATE_TABLE_SQL="CREATE TABLE
boraji.users (" + "UID INT NOT NULL," + "NAME VARCHAR(45) NOT NULL," +
"DOB DATE NOT NULL," + "EMAIL VARCHAR(45) NOT NULL," + "PRIMARY KEY
(UID))";
…
stmt = conn.createStatement();
stmt.executeUpdate(CREATE_TABLE_SQL);
```

2.5 Ejecución de consultas

Como es sabido, para realizar consultas sobre bases de datos se utiliza la sentencia SELECT. Según se vio en el apartado anterior, el método encargado de ejecutar las sentencias SELECT es executeQuery(), perteneciente a la interfaz java.sql.ResultSet:

```
public ResultSet executeQuery (String sql) throws SQLException
```

Al ejecutar una consulta del tipo SELECT, **se devuelve** un objeto Statement de la interfaz java.sql.**ResultSet**, que almacena el conjunto de resultados producidos por la sentencia ejecutada. Además, esta interfaz contiene los métodos necesarios para moverse por dichos registros que se obtienen al ejecutar la consulta SELECT, así como los métodos para recuperar los datos que componen cada uno de estos registros.

El método fundamental para moverse por las distintas filas obtenidas por la consulta es **next()**, que avanzará uno a uno en los registros devueltos por la consulta. Al ejecutarse la primera vez, el ResultSet apuntará a la primera fila de los resultados, al ejecutarse de nuevo, a la segunda; y así sucesivamente hasta llegar al final, en cuyo caso devolverá false.

Por tanto, es muy sencillo recorrer todos los resultados obtenidos por la consulta mediante un bucle while, de la siguiente forma:

```
while (rs.next()) {
}
```

Siendo *rs* el ResultSet devuelto por la consulta.

En cada una de las iteraciones de este bucle, se podrá acceder a la información contenida en los campos de la fila actual. Para ello, se puede hacer uso de cualquiera de los siguientes métodos, según el tipo del dato:

- getBoolean().
- getInt().
- getShort().
- getByte().
- getDate().
- getDouble().
- getFloat().

La sintaxis de, por ejemplo, el método getString, sería la siguiente:

```
public String getString(int columnIndex) throws SQLException
public String getString(String columnName) throws SQLException
```

Como se puede ver, se dispone de dos métodos con diferentes parámetros de entrada. Ocurre igual para el resto de los métodos mencionados, que se diferenciarán de este en el tipo de los valores devueltos. Así, pues, se les puede pasar como parámetro, tanto el nombre de la columna (tipo String) como el índice de la columna (tipo int). Se debe tener en cuenta que el índice de la primera columna es 1.

EJEMPLO PRÁCTICO

En el proyecto que estamos trabajando dentro de nuestra empresa, necesitamos tener un método para poder listar todos los vehículos que hemos insertado en la base de datos. Ese método se incluirá dentro de la misma clase que previamente tenía ya métodos para el establecimiento de la conexión.

¿Cómo realizamos dicho método teniendo en cuenta que el método debe devolver el ResultSet que será tratado fuera de la clase?

En este caso necesitamos un nuevo método que realice la siguiente operación y tenga el siguiente interfaz.

```
public ResultSet listarVehiculos() throws SQLException {
     String sqlString = "SELECT * FROM VEHICULOS ";
       Statement statement = connection.createStatement();
       ResultSet rs = statement.executeQuery(sqlString);
       return rs;
}
```

Este ejemplo, implementa un método llamado listarVehiculos() para recuperar todos los datos de la tabla empleados.

- En primer lugar, este método crea un objeto Statement, utilizado para invocar el método executeQuery() con una instrucción SQL (SELECT) como argumento.

- El objeto java.sql.ResultSet contiene todas las filas de la tabla VEHICULOS que coinciden con la instrucción SELECT.

Además de lo estudiado, la interfaz ResultSet permite conocer la estructura de los resultados obtenidos y, a partir de esta, obtener más información a través de los siguientes métodos:

- getTableName().
- getMetaData().
- getColumnCount().
- getColumnName().
- getColumnType().

Por ejemplo, dado un bloque de resultados, se puede utilizar el método getColumnCount() para obtener el número de columnas y, en base a esto, obtener la información de tipo asociada a cada una de ellas.

PARA SABER MÁS

Accede para encontrar información sobre el objeto ResultSet y ejemplos de uso:

VÍDEO DE INTERÉS

Visualiza un ejemplo sobre Statements y ResultSet:

2.6 Ejecución de sentencias de manipulación de datos

En el caso de manipulación de datos, que no estén relacionadas con la manipulación de definición de datos, tenemos dos instrucciones, principalmente en el lenguaje SQL:

- **INSERT:** permite insertar nuevas filas y registros en una tabla. Su estructura tiene tres partes: la tabla a la que referenciamos, las columnas implicadas y sus valores.

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

- **UPDATE**: permite actualizar datos y filas de una base de datos. En la estructura volvemos a encontrarnos con los objetos, tabla, columnas y valores. En este caso, es necesario también indicar la condición sobre la que seleccionaremos las filas o los objetos implicados.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Independientemente de la operación de modificación que queramos realizar sobre la base de datos, lo primero que se debe hacer es preparar la sentencia que se desea ejecutar mediante un **PrepareStatement("String sentencia")**, y los pasos a seguir serían:

1. Crear la conexión.
2. Crear un Statement con el SQL UPDATE a realizar usando el método PreparedStatement.
3. Colocar los datos a actualizar.

4. Ejecutar la sentencia.
5. Cerrar la conexión.

Veamos un ejemplo con código Java:

```
        Connection conn = DriverManager.getConnection(myUrl, "root",
"");

        // create the java mysql update preparedstatement
        String query = "update users set num_points = ? where first_name
= ?";
        PreparedStatement preparedStmt = conn.prepareStatement(query);
        preparedStmt.setInt    (1, 6000);
        preparedStmt.setString(2, "Fred");

        // execute the java preparedstatement
        preparedStmt.executeUpdate();

        conn.close();
```

VÍDEO DE INTERÉS

Profundiza sobre PreparedStatement para saber cómo se utiliza en JDBC y saber cómo proporcionarle parámetros:

EJEMPLO PRÁCTICO

En el proyecto que estamos trabajando, los usuarios tendrán un interfaz en forma de formulario en el que introducirán los datos para que sean grabados en la base de datos.

Uno de los interfaces consiste en dar de alta un nuevo vehículo para lo que se necesitan los siguientes datos:

- Matricula.
- Tipo de vehículo.
- Marca de vehículo.
- idDepartamento.

¿Cómo realizaríamos un método dentro de una clase DB que ya tiene otros métodos para realizar la conexión y, por lo tanto, sólo necesitamos la inserción?

En este caso, sólo necesitamos añadir un nuevo método a esa clase que tenga el siguiente código:

```
private void nuevoVehiculo(String matricula, String tipo, String
marca, int dep) throws SQLException {
stmt = con.prepareStatement("INSERT INTO VEHICULOS VALUES
(?,?,?,?)");
stmt.setString(1, matricula);
stmt.setString(2, tipo);
stmt.setString(3, marca);
stmt.setInt(4, dep);
stmt.executeUpdate();

}
```

2.7 Ejecución de procedimientos almacenados en la base de datos

Un procedimiento almacenado es una función que se encuentra alojada en la propia base de datos. Su principal característica es que devuelve uno o más parámetros.

Los procedimientos almacenados más sencillos no requieren parámetros de entrada y solo devuelven un conjunto de datos de salida, como si se tratara de una consulta SQL.

Para ejecutar procedimientos almacenados desde una aplicación desarrollada con el lenguaje de programación Java, primero es necesario cargar el procedimiento en la base de datos.

PARA SABER MÁS

Aprende cómo crear procedimientos almacenados:

Una vez realizada la carga, para ejecutarlo desde la aplicación Java, se usa el método prepareCall(nombre_funcion). Este devuelve un objeto CallableStatement, que permite ejecutar procedimientos almacenados en bases de datos. Evidentemente, la conexión con la base de datos se tiene que haber establecido previamente, a través de la interfaz DriverManager, tal como se vio en apartados anteriores.

Es importante recordar que hay que realizar la gestión de excepciones a través de try-catch-finally.

Los parámetros de entrada se establecen utilizando los métodos set<Tipo>(), donde <Tipo> es el tipo de dato correspondiente, los parámetros de salida se registran utilizando el método registerOutParameter. Después de ejecutar el procedimiento almacenado con execute, los valores de los parámetros de salida se recuperan con los métodos get<Tipo>(), similares a los utilizados para parámetros de entrada.

Por ejemplo, si suponemos que tenemos el siguiente procedimiento para obtener el salario de los empleados:

```
DELIMITER //
CREATE PROCEDURE obtenerSalario(IN empleadoId INT, OUT salario
DECIMAL(10,2))
BEGIN
    SELECT salario INTO salario
    FROM empleados
    WHERE id = empleadoId;
END //
DELIMITER ;
```

Vemos que tenemos 2 parámetros, 1 de entrada (el ID del empleado) y uno de salida que es el salario que tiene el empleado. Podemos hacer uso de los parámetros con el siguiente código en Java.

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Types;

public class ProcedimientoAlmacenado {

      final static String url = "jdbc:mysql://localhost:3306/empresa";
    final static String usuario = "root";
    final static String contraseña = "";

    public static void main(String[] args) {
        Connection conexion = null;
        CallableStatement callableStatement = null;

        try {
            // Establece la conexión
            conexion = DriverManager.getConnection(url, usuario,
contraseña);

            // Preparar la llamada al procedimiento almacenado
            callableStatement = conexion.prepareCall("{call
obtenerSalario(?, ?)}");

            // Establecer el parámetro de entrada
            callableStatement.setInt(1, 1); // Suponiendo que el ID
del empleado es 1

            // Registrar el parámetro de salida
            callableStatement.registerOutParameter(2, Types.DECIMAL);

            // Ejecutar el procedimiento almacenado
            callableStatement.execute();

            // Obtener el valor del parámetro de salida
            BigDecimal salario = callableStatement.getBigDecimal(2);

            System.out.println("El salario del empleado es: " +
salario);

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (callableStatement != null)
callableStatement.close();
                if (conexion != null) conexion.close();
```

```
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Cabe destacar que el primer parámetro queda marcado como el parámetro 1 y no como el parámetro 0 como estamos acostumbrados en la programación convencional. Si ejecutáramos este código en una base de datos con el procedimiento almacenado este nos devolvería el salario del empleado de código 1.

VÍDEO DE INTERÉS

Comprueba cómo crear procedimientos almacenados en MySQL:

2.8 Gestión de transacciones

Una de las principales características de las bases de datos es la capacidad para soportar transacciones. Esto permite que si hay algún problema mientras se escribe en la base de datos, ésta volverá al estado en el que se encontraba antes de la escritura de la información, es decir, permite que se deshagan operaciones que se cree que no se han realizado correctamente, debido, por ejemplo, a que el sistema operativo ha caído.

Si una transacción incluye varias instrucciones, éstas se realizarán al completo de manera exitosa o todas deberán ser descartadas.

La API JDBC, mediante la clase Connection, da soporte a la gestión de transacciones, de forma que es posible deshacer un conjunto de operaciones relacionadas si es necesario.

Cuando se crea un objeto Connection, se configura para realizar automáticamente cada una de las transacciones, de forma que cuando se ejecuta una sentencia, ésta se ejecuta en el mismo momento en que se solicita. Para modificar este comportamiento se debe hacer uso de los siguientes métodos de la interfaz Connection:

- **Método setAutoCommit(boolean autoCommit**): otorga el control sobre lo que se está realizando y cuándo se está realizando. Por defecto, es true, es decir, cada instrucción se ejecuta de forma inmediata cuando se solicita. Por tanto, si se quieren agrupar varias instrucciones como parte de una sola transacción, será necesario poner este valor a false antes de iniciar la transacción.

- **Método commit():** ejecuta todas las instrucciones desde el último commit.

- **Método rollback():** deshace los cambios realizados desde el último commit.

PARA SABER MÁS

Amplía información sobre la gestión de transacciones en Java:

ENLACE DE INTERÉS

Revisa este proyecto creado para gestionar una biblioteca con MySQL y JDBC:

2.9 Configuración de un Proyecto con MySQL

El primer paso consiste en configurar la base de datos MySQL, si no tenemos una base de datos MySQL hay que instalarla, para ello existen muchos proveedores como el propio MySQL o XAMPP/WAMP.

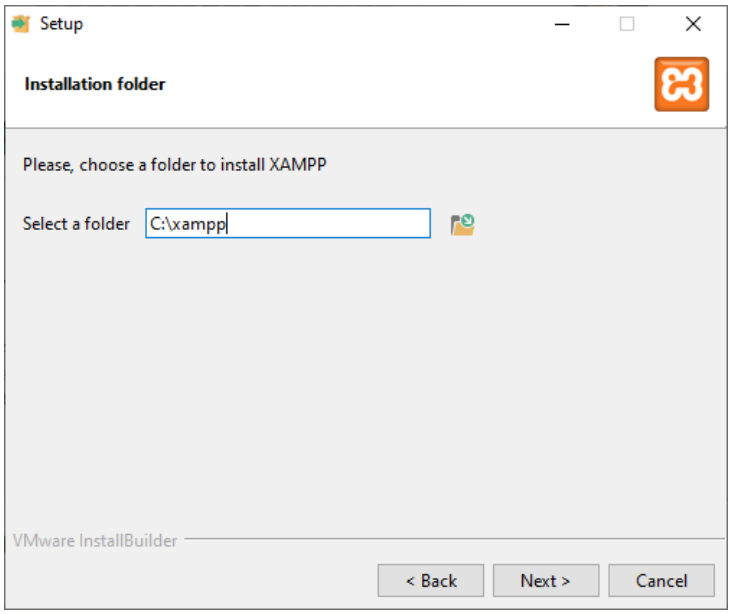
En este caso vamos a utilizar XAMPP para instalarlo ya que viene con muchas configuraciones por defecto que nos facilitará la tarea de configuración. Para instalarlo, debemos acceder a la página oficial.

Descarga XAMPP
Fuente: Elaboración propia

ENLACE DE INTERÉS

Aquí podrás descargar XAMPP:

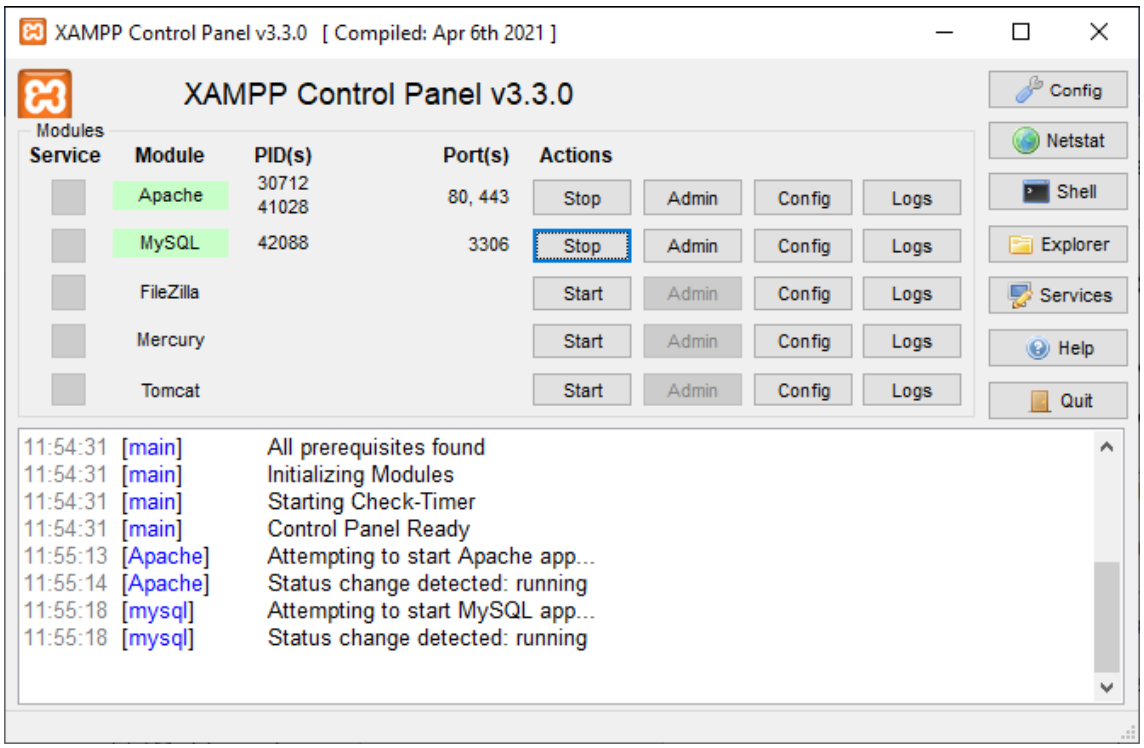
Si observamos, pone MariaDB en vez MySQL, esto se debe a que MariaDB es una variante de MySQL completamente libre. Una vez se abra el instalador pulsaremos “Next >” hasta la selección de carpeta donde deberemos decidir donde instalarlo, en mi caso lo dejaré en su ruta por defecto.



Instalación XAMPP

Fuente: Elaboración propia

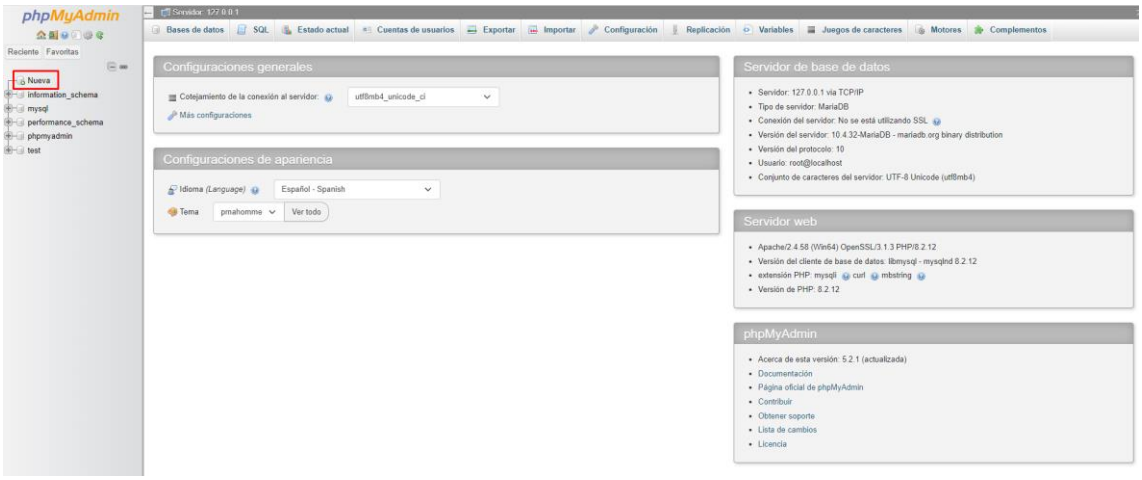
A partir de ahí. pulsaremos “Next >” hasta llegar a su instalación. Cuando terminemos nos saldrá un panel, en el que pulsaremos en el botón “Start” en Apache y en MySQL.



Panel de Control XAMPP

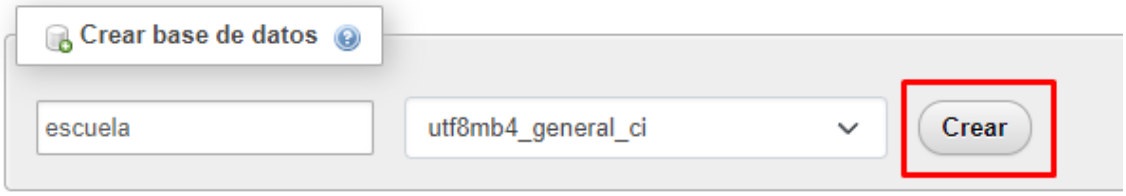
Fuente: Elaboración propia

Si escribimos en el navegador http://localhost/phpmyadmin/ podremos acceder a la interfaz de MySQL. En esta interfaz pulsaremos en “Nueva” para crear una BBDD.



Interfaz MySQL phpMyAdmin
Fuente: Elaboración propia

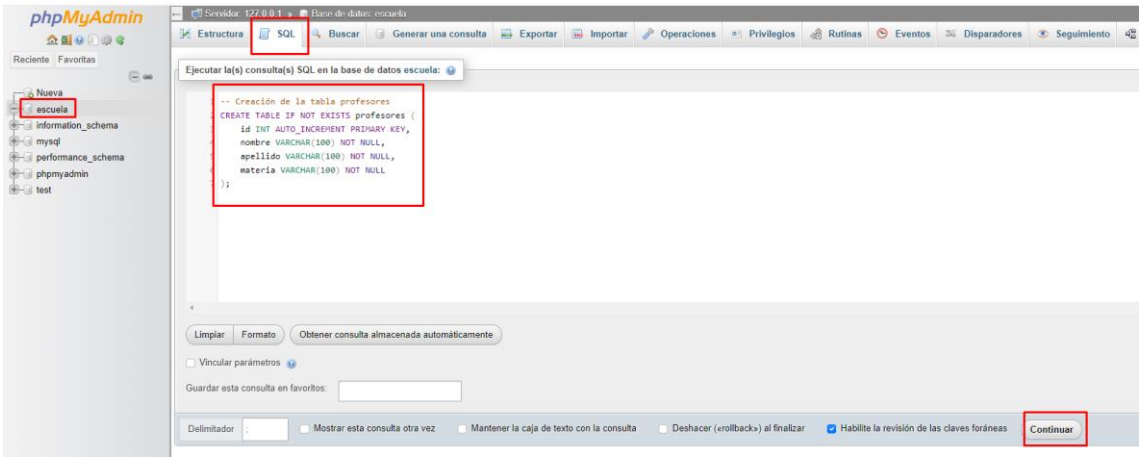
Ahora escribimos escuela y pulsamos en Crear.



Creación BBDD
Fuente: Elaboración propia

Lanzaremos un script en SQL para crear la base de datos (esto se podría hacer desde el código en Java, pero vamos a hacerlo desde la interfaz).

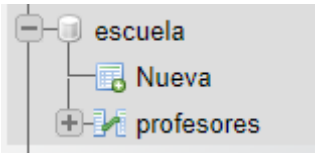
```sql
-- Creación de la tabla profesores
CREATE TABLE IF NOT EXISTS profesores (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    apellido VARCHAR(100) NOT NULL,
    materia VARCHAR(100) NOT NULL
);
```



Ejecución de consultas phpMyAdmin

Fuente: Elaboración propia

Ahora, si desplegamos “escuela”, encontraremos la tabla de profesores.



Tablas BBDD

Fuente: Elaboración propia

Pasaremos a la parte en Java. Para ello vamos a crear un proyecto escuela e importaremos el JDBC como se explica en el apartado 2.2. En este proyecto crearemos una clase llamada “Principal”. Pondremos el siguiente código.

```
package escuela;

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

public class Principal {

    private static final String URL =
"jdbc:mysql://localhost:3306/escuela";
    private static final String USER = "root"; // Cambia por tu
usuario de MySQL
    private static final String PASSWORD = ""; // Cambia por tu
contraseña de MySQL

    public static void main(String[] args) {
        try {
            // Conexión a la base de datos
```

```
            Connection conexion = DriverManager.getConnection(URL,
USER, PASSWORD);
            System.out.println("Conexión exitosa a la base de
datos.");

            // Inserción de registros
            String insertarSQL = "INSERT INTO profesores (nombre,
apellido, materia) VALUES (?, ?, ?)";
            PreparedStatement ps =
conexion.prepareStatement(insertarSQL);

            // Primer profesor
            ps.setString(1, "Juan");
            ps.setString(2, "Pérez");
            ps.setString(3, "Matemáticas");
            ps.executeUpdate();

            // Segundo profesor
            ps.setString(1, "Ana");
            ps.setString(2, "García");
            ps.setString(3, "Historia");
            ps.executeUpdate();

            // Tercer profesor
            ps.setString(1, "Luis");
            ps.setString(2, "Martínez");
            ps.setString(3, "Ciencias");
            ps.executeUpdate();

            System.out.println("Registros insertados correctamente.");

            // Consulta y muestra los registros
            String consultaSQL = "SELECT * FROM profesores";
            Statement stmt = conexion.createStatement();
            ResultSet rs = stmt.executeQuery(consultaSQL);

            System.out.println("Lista de profesores:");
            while (rs.next()) {
                System.out.println("ID: " + rs.getInt("id") +
                                   ", Nombre: " +
rs.getString("nombre") +
                                   ", Apellido: " +
rs.getString("apellido") +
                                   ", Materia: " +
rs.getString("materia"));
            }

            // Cerrar conexiones
            rs.close();
            stmt.close();
            ps.close();
            conexion.close();
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

Si ejecutamos el código, el resultado será el siguiente.

```
Conexión exitosa a la base de datos.
Registros insertados correctamente.
Lista de profesores:
ID: 1, Nombre: Juan, Apellido: Pérez, Materia: Matemáticas
ID: 2, Nombre: Ana, Apellido: García, Materia: Historia
ID: 3, Nombre: Luis, Apellido: Martínez, Materia: Ciencias
```

Resultado Ejecución

Fuente: Elaboración propia

Y en la base de datos también podemos verlo.

		id	nombre	apellido	materia
☐	Editar Copiar Borrar	1	Juan	Pérez	Matemáticas
☐	Editar Copiar Borrar	2	Ana	García	Historia
☐	Editar Copiar Borrar	3	Luis	Martínez	Ciencias

Tabla Profesores

Fuente: Elaboración propia

EJEMPLO PRÁCTICO

En nuestro proyecto para la gestión de vehículos, necesitamos un método que gestione las fechas de las ITVs de los vehículos al mismo tiempo que se actualizan la tabla del vehículo. Los pasos que se necesitan realizar son:

- Insertar nueva fecha planificada ITV.
  - Matrícula.
  - FechaPrevista.
- Actualizar campo itvPlanificado en vehículos.

¿Cómo optimizar estas dos acciones en un solo método?

Crearemos un método que ejecute una transacción para realizar de forma atómica ambas acciones, y de esa forma no haya incoherencias en la base de datos.

```
try{
     // Iniciar una transacción
     connection.setAutoCommit(false);

     Statement statement = connection.createStatement();

     // Crear una nueva itv
     statement.executeUpdate( "INSERT INTO ITV(MATRICULA" +
         "FECHA_PREVISTA,...) VALUES(...)'');

     // Actualizar el paqrque de vehiculos
     statement.executeUpdate("UPDATE TABLE VEHICULOS " +
"SET itvPlanificado = TRUE" +
"WHERE MATRICULA = ...'');

     // Crear un registro de envíos
     if (...) {
          // Operación éxitosa
          connection.commit();
     } else {
          // Deshacer operación
          connection.rollback();
     }
} catch (SQLException) {
     // Manejar excepciones aquí
} finally {
     // Cerrar instrucción y conexión
}
```

RESUMEN FINAL

Un sistema gestor de base de datos (**SGBD**) es un conjunto de programas cuya función es administrar y gestionar la información contenida una base de datos.

Las bases de datos y los sistemas de gestión suelen tener su propio protocolo de comunicación y, por lo tanto, la necesidad de conectores específicos para la comunicación con las bases de datos es imprescindible. En el lenguaje Java tenemos cuatro modelos de conectores, siendo los más famosos y usados los conectores JDBC nativos.

Estos conectores específicos nos permiten usar ya protocolos estándar de gestión de bases de datos para todas las operaciones necesarias: definición de datos, consulta de información y modificación de la información.

Para ello, el proceso de trabajo es siempre el mismo, incorporamos al proyecto el conector adecuado, gestionamos la conexión, realizamos las sentencias y cerramos la conexión.

En esta simbiosis entre sistema de gestión de base de datos y lenguaje de programación, debemos aprovechar todas las herramientas que nos ofrecen ambas partes para tener un proyecto lo más escalable posible, usando herramientas como procedimientos almacenados de los SGBD que nos permitirán reutilizarlas en otros proyectos.