

## TEMA13.

### ACCESO A BASES DE DATOS

Programación  
CFGS DAW

Profesores:

Carlos Cacho

Raquel Torres

[carlos.cacho@ceedcv.es](mailto:carlos.cacho@ceedcv.es)

[raquel.torres@ceedcv.es](mailto:raquel.torres@ceedcv.es)

Versión:180427.2028

## Licencia



**Reconocimiento - NoComercial - CompartirIgual (by-nc-sa):** No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

## Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:



Importante



Atención



Interesante

# ÍNDICE DE CONTENIDO

<b>1. Conceptos básicos.....</b>	<b>4</b>
1.1 El lenguaje SQL.....	5
1.1.1 Comandos.....	5
1.1.2 Clausulas.....	6
1.1.3 Operadores.....	6
1.1.4 Ejemplos.....	7
<b>2. Java y los SGBD.....</b>	<b>9</b>
2.1 Funciones del JDBC.....	9
2.2 Drivers JDBC.....	10
<b>3. Controlador MySQL/MariaDB.....</b>	<b>10</b>
3.1 Instalación del Driver.....	11
3.2 Conexión a través del driver JDBC.....	13
3.3 Conectar con la Base de Datos.....	16
3.4 Crear una tabla.....	17
3.4.1 A través de sentencias SQL.....	17
3.4.2 A través del asistente de creación de tablas.....	18
3.5 Insertar datos en una tabla.....	19
3.6 Consultar valores.....	19
<b>4. Conexión a una base de datos.....</b>	<b>20</b>
<b>5. Ejecución de consultas.....</b>	<b>22</b>
5.1 Obteniendo datos del <i>ResultSet</i> .....	24
5.2 Tipos de datos y conversiones.....	24
5.3 Desplazamiento en un <i>ResultSet</i> .....	25
<b>6. Modificación.....</b>	<b>26</b>
<b>7. Inserción y Borrado.....</b>	<b>27</b>
<b>8. Ejemplo.....</b>	<b>29</b>
<b>9. Agradecimientos.....</b>	<b>32</b>

## UD13. ACCESO A BASES DE DATOS

### 1. CONCEPTOS BÁSICOS



Una **base de datos** es una colección de datos clasificados y estructurados que son guardados en uno o varios ficheros, pero referenciados como si de un único fichero se tratara.

Para crear y manipular bases de datos relacionales, existen en el mercado varios sistemas administradores de bases de datos; por ejemplo, Access, SQL Server, Oracle y DB2. Otros sistemas administradores de bases de datos de libre distribución son MySQL/MariaDB y PostgreSQL.

Los datos de una base de datos relacional se almacenan en tablas lógicamente relacionadas entre sí utilizando campos clave comunes. A su vez, cada tabla dispone los datos en filas y columnas. Por ejemplo, piensa en una relación de clientes, a las filas se les denomina **tuplas** o **registros** y a las columnas **campos**.

Los usuarios de un sistema administrador de bases de datos pueden realizar sobre una determinada base operaciones como insertar, recuperar, modificar y eliminar datos, así como añadir nuevas tablas o eliminarlas. Estas operaciones se expresan generalmente en un lenguaje denominado **SQL**.

Por lo tanto antes de empezar debemos descargarnos el software necesario para poder tener un servidor de base de datos, por ejemplo **XAMPP (Window/Linux Apache MariaDB PHP Perl)** que incorpora un servidor web y la herramienta phpMyAdmin para trabajar con bases de datos.

Si no lo tienes instalado porque no cursas el módulo de BBDD, lo puedes descargar desde la dirección: <https://www.apachefriends.org/es/index.html>

Veréis que las versiones más actuales de XAMPP han sustituido MySQL por MariaDB como SGBD (Sistema Gestor de Bases de Datos). No pasa nada, ambos SGBD son compatibles y pueden utilizarse indistintamente. Para más información sobre las diferencias entre MySQL y MariaDB podéis leer el siguiente [artículo](#).



Para su instalación, tan solo debes seguir los pasos del instalador.

## 1.1 El lenguaje SQL

SQL incluye operaciones tanto de definición, por ejemplo CREATE, como de manipulación de datos, por ejemplo INSERT, UPDATE, DELETE y SELECT.

El lenguaje SQL está distribuido en cuatro grandes bloques, que se combinan para crear, actualizar y manipular las bases de datos, estos bloques son:

- Comandos
- Cláusulas
- Operadores
- Funciones

### 1.1.1 Comandos

Existen tres tipos de comandos en SQL:

- Los DDL (Data Definition Language), que permiten crear, modificar y borrar nuevas bases de datos, tablas, campos y vistas.
- Los DML (Data Manipulation Language), que permiten introducir información en la BD, borrarla y modificarla.
- Los DQL (Data Query Language), que permiten generar consultas para ordenar, filtrar y extraer información de la base de datos.

Los comandos DDL son:

- **CREATE**: Crear nuevas tablas, campos e índices.
- **ALTER**: Modificación de tablas añadiendo campos o modificando la definición de los campos.

- *DROP*: Instrucción para eliminar tablas, campos e índices.

Los comandos DML son:

- *INSERT*: Insertar registros a la base de datos.
- *UPDATE*: Instrucción que modifica los valores de los campos y registros especificados en los criterios.
- *DELETE*: Eliminar registros de una tabla de la base de datos.

El principal comando DQL es:

- *SELECT*: Consulta de registros de la base de datos que cumplen un criterio determinado.

### 1.1.2 Clausulas

Las cláusulas son condiciones de modificación, utilizadas para definir los datos que se desean seleccionar o manipular:

- *FROM*: Utilizada para especificar la tabla de la que se seleccionarán los registros.
- *WHERE*: Cláusula para detallar las condiciones que deben reunir los registros resultantes.
- *GROUP BY*: Utilizado para separar registros seleccionados en grupos específicos.
- *HAVING*: Utilizada para expresar la condición que ha de cumplir cada grupo.
- *ORDER BY*: Utilizada para ordenar los registros seleccionados de acuerdo a una ordenación específica.
- 

### 1.1.3 Operadores

Operadores lógicos:

- *AND*: Evalúa dos condiciones y devuelve el valor cierto, si ambas condiciones son ciertas.
- *OR*: Evalúa dos condiciones y devuelve el valor cierto, si alguna de las dos condiciones es cierta.
- *NOT*: Negación lógica. Devuelve el valor contrario a la expresión.

Operadores de comparación:

- *< [...]* menor que [...]
- *> [...]* mayor que [...]
- *<> [...]* diferente a [...]
- *<= [...]* menor o igual que [...]
- *>= [...]* mayor o igual que [...]
- *= [...]* igual que [...]
- *BETWEEN*: Especifica un intervalo de valores
- *LIKE*: Compara un modelo
- *IN*: Operadores para especificar registros de una tabla

### 1.1.4 Ejemplos

Para crear una base de datos, SQL proporciona la sentencia:

```
CREATE DATABASE <base de datos>
```

Para eliminar una base de datos, se ejecuta la sentencia:

```
DROP DATABASE <base de datos>
```

Para crear una tabla, SQL proporciona la sentencia:

```
CREATE TABLE <tabla> ( <columna 1> [,<columna 2>] ...)
```

donde <columna n> se formula según la siguiente sintaxis:

```
<columna n> <tipo de dato> [DEFAULT <expresion>] [<const 1> [<const2>]...]
```

La cláusula *DEFAULT* permite especificar un valor por omisión para la columna y, opcionalmente, para indicar la forma o característica de cada columna, se pueden utilizar las constantes: *NOT NULL*, *UNIQUE* o *PRIMARY KEY*.

La cláusula *PRIMARY KEY* se utiliza para definir la columna como clave principal de la tabla. Esto supone que la columna no puede contener valores nulos ni duplicados. Una tabla puede contener una sola restricción *PRIMARY KEY*.

La cláusula *UNIQUE* indica que la columna no permite valores duplicados. Una tabla puede tener varias restricciones *UNIQUE*.

Por ejemplo:

```
CREATE TABLE telefonos(  
nombre CHAR(30) NOT NULL,  
direccion CHAR(30) NOT NULL,  
telefono CHAR(12) PRIMARY KEY NOT NULL,  
observaciones CHAR(240)  
)
```

Para escribir datos en una tabla, SQL proporciona la sentencia:

```
INSERT [INTO] <tabla> [( <columna 1> [,<columna 2>] ...)]  
VALUES (<expresion 1> [,<expresion 2>] ...),...
```

Por ejemplo:

```
INSERT INTO telefonos VALUES ('Pepito Pérez','VALENCIA','12345678','Nada')
```

Para modificar datos en una tabla, SQL proporciona la sentencia:

```
UPDATE <tabla> SET <columna1> = (<expresion1> | NULL)
[<columna2> = (<expresion2> | NULL)]...
WHERE <condición de búsqueda>
```

Por ejemplo:

```
UPDATE telefonos SET direccion = "Puerto de Sagunto"
WHERE telefono = '12345678'
```

Para borrar registros en una tabla, SQL proporciona la sentencia:

```
DELETE FROM <tabla> WHERE <condición de búsqueda>
```

Por ejemplo:

```
DELETE FROM telefonos WHERE telefono='12345678'
```

Para borrar una tabla de la BBDD, se ejecuta la siguiente sentencia:

```
DROP TABLE <tabla>
```

Para seleccionar datos de una tabla, SQL proporciona la sentencia:

```
SELECT [ALL | DISTINCT] <lista de selección>
FROM <tablas>
WHERE <condiciones de selección>
[ORDER BY <columna1> [ASC|DESC][, <columna2>[ASC|DESC]]...]
```



Por ejemplo:

```
SELECT * FROM telefonos;  
SELECT * FROM telefonos ORDER BY nombre;  
SELECT * FROM telefonos WHERE telefono > '1234';  
SELECT * FROM telefonos WHERE telefono LIKE '91*';
```

## 2. JAVA Y LOS SGBD

Java puede conectarse con distintos SGBD y en diferentes sistemas operativos. Independientemente del método en que se almacenen los datos debe existir siempre un **mediador** entre la aplicación y el sistema de base de datos y en Java esa función la realiza **JDBC**.



Para la conexión a las bases de datos utilizaremos la API estándar de JAVA denominada **JDBC** (Java Data Base Connection)

JDBC es un API incluido dentro del lenguaje Java para el acceso a bases de datos. Consiste en un conjunto de clases e interfaces escritas en Java que ofrecen un completo API para la programación con bases de datos, por lo tanto es la única solución 100% Java que permite el acceso a bases de datos.

JDBC es una especificación formada por una colección de interfaces y clases abstractas, que todos los fabricantes de drivers deben implementar si quieren realizar una implementación de su driver 100% Java y compatible con JDBC (JDBC-compliant driver). Debido a que JDBC está escrito completamente en Java también posee la ventaja de ser independiente de la plataforma.



No será necesario escribir un programa para cada tipo de base de datos, una misma aplicación escrita utilizando JDBC podrá manejar bases de datos Oracle, Sybase, SQL Server, etc.

Además podrá ejecutarse en cualquier sistema operativo que posea una Máquina Virtual de Java, es decir, serán aplicaciones completamente independientes de la plataforma. Otras APIs que se suelen utilizar bastante para el acceso a bases de datos son DAO (Data Access Objects) y RDO (Remote Data Objects), y ADO (ActiveX Data Objects), pero el problema que ofrecen estas soluciones es que sólo son para plataformas Windows.

JDBC tiene sus clases en el paquete *java.sql* y otras extensiones en el paquete *javax.sql*.

## 2.1 Funciones del JDBC

Básicamente el API JDBC hace posible la realización de las siguientes tareas:

- Establecer una conexión con una base de datos.
- Enviar sentencias SQL.
- Manipular los datos.
- Procesar los resultados de la ejecución de las sentencias.

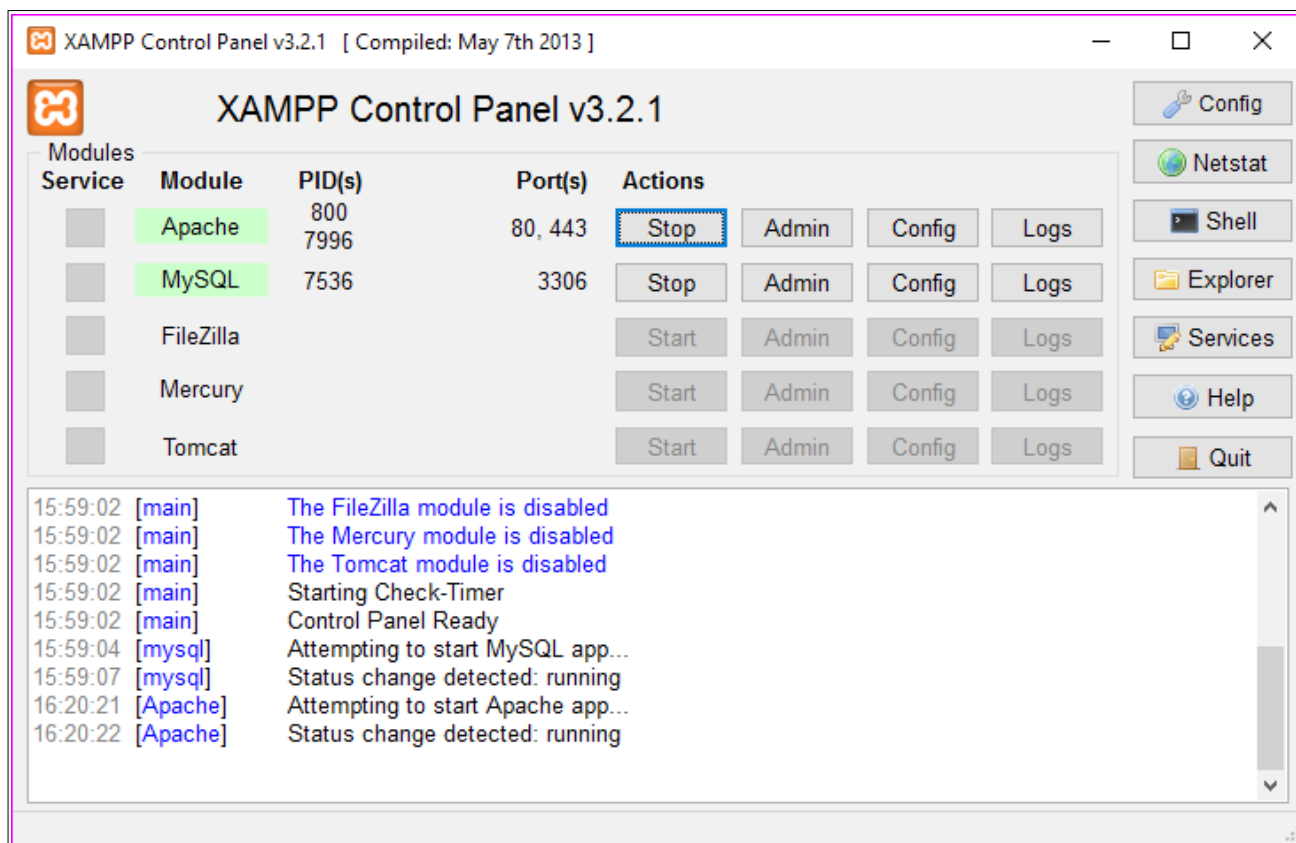
## 2.2 Drivers JDBC

Los drivers nos permiten conectarnos con una base de datos determinada. Existen **cuatro tipos de drivers JDBC**, cada tipo presenta una filosofía de trabajo diferente, a continuación se pasa a comentar cada uno de los drivers:

- JDBC-ODBC bridge plus ODBC driver (tipo 1): permite al programador acceder a fuentes de datos ODBC existentes mediante JDBC. El JDBC-ODBC Bridge (puente JDBC-ODBC) implementa operaciones JDBC traduciéndolas a operaciones ODBC, se encuentra dentro del paquete *sun.jdbc.odbc* y contiene librerías nativas para acceder a ODBC.  
Al ser usuario de ODBC depende de las dll de ODBC y eso limita la cantidad de plataformas en donde se puede ejecutar la aplicación.
- Native-API partly-Java driver (tipo 2): son similares a los drivers de tipo1, en tanto en cuanto también necesitan una configuración en la máquina cliente. Este tipo de driver convierte llamadas JDBC a llamadas de Oracle, Sybase, Informix, DB2 u otros SGBD. Tampoco se pueden utilizar dentro de applets al poseer código nativo.
- JDBC-Net pure Java driver (tipo 3): Estos controladores están escritos en Java y se encargan de convertir las llamadas JDBC a un protocolo independiente de la base de datos y en la aplicación servidora utilizan las funciones nativas del sistema de gestión de base de datos mediante el uso de una biblioteca JDBC en el servidor. La ventaja de esta opción es la portabilidad.
- JDBC de Java cliente (tipo 4): Estos controladores están escritos en Java y se encargan de convertir las llamadas JDBC a un protocolo independiente de la base de datos y en la aplicación servidora utilizan las funciones nativas del sistema de gestión de base de datos sin necesidad de bibliotecas. La ventaja de esta opción es la portabilidad. Son como los drivers de tipo 3 pero sin la figura del intermediario y tampoco requieren ninguna configuración en la máquina cliente. Los drivers de tipo 4 se pueden utilizar para servidores Web de tamaño pequeño y medio, así como para intranets.

## 3. CONTROLADOR MYSQL/MARIADB

Siempre que queramos trabajar con el servidor y las bases de datos del mismo deberemos iniciar la herramienta XAMPP descargada anteriormente y arrancar los servicios Apache y MySQL/MariaDB:

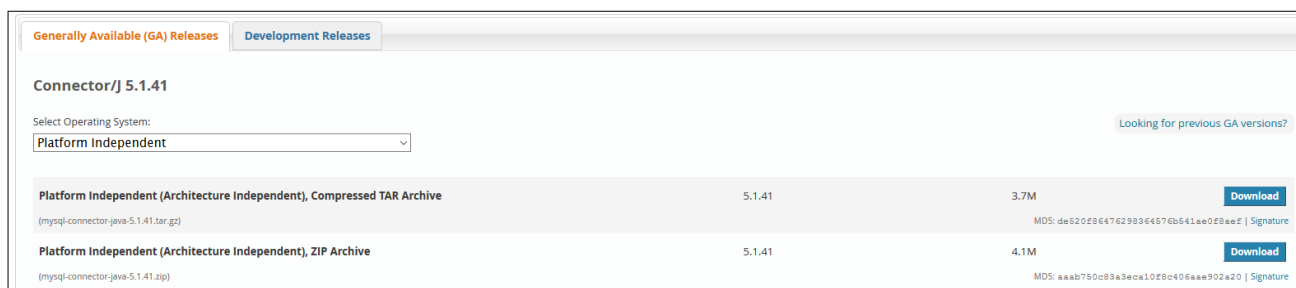


### 3.1 Instalación del Driver

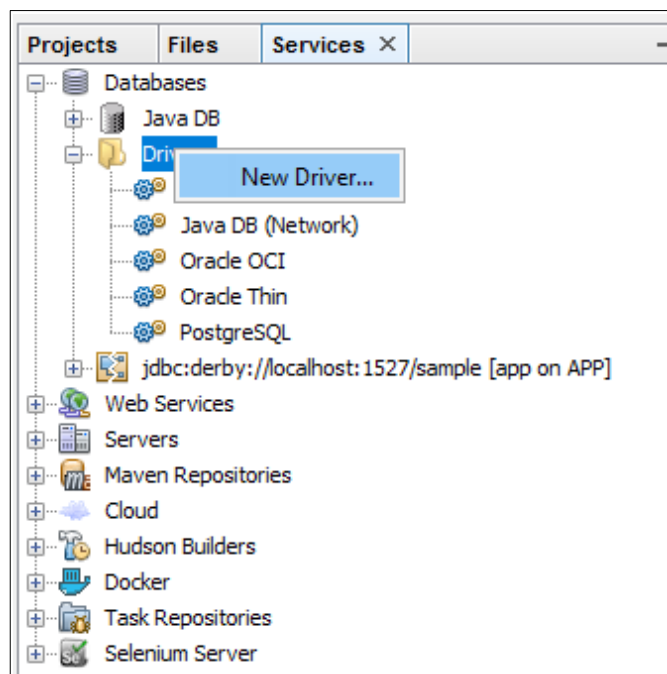
Para poder utilizar el explorador de bases de datos del IDE NetBeans con una base de datos MySQL/MariaDB, hay que poner a disposición del IDE el controlador MySQL/MariaDB. Para ello, haz clic con el botón derecho del ratón en la opción *Drivers* del panel *Service* y ejecuta la orden *Add Driver* del menú contextual que se visualiza. Se muestra el diálogo *add JDBC Driver*.

El driver lo podemos descargar de la dirección:

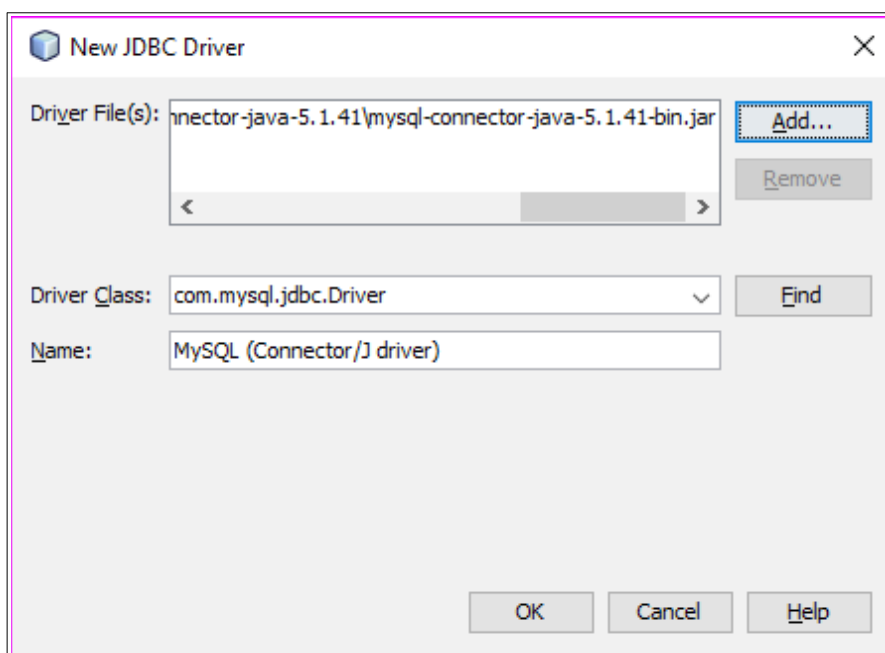
<http://dev.mysql.com/downloads/connector/j/>



No hace falta registrarse para descargarlo, solo hay pulsar en “*No thanks, just start my download.*”



Descomprimos el .zip y nos debe de generar un fichero llamado mysql-connector-java-5.1.41-bin.jar (puede ser esta versión o alguna posterior). Le damos al botón de Add... y seleccionamos el fichero mysql-connector-java-5.1.41-bin.jar (que previamente lo habremos descomprimido en alguna carpeta de nuestro sistema). Una vez seleccionado el fichero nos debe de rellenar la clase principal del Driver que en nuestro caso debe de ser com.mysql.jdbc.Driver y el nombre a nuestro driver, por ejemplo MySQL (Connector/J driver).

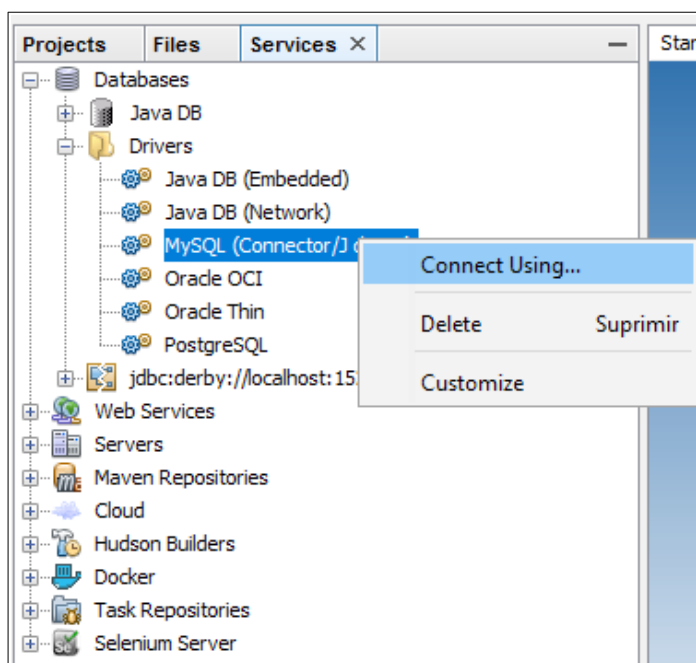


Llegados a este punto ya tenemos enlazado el driver JDBC de MySQL/MariaDB desde el Database Explorer, ahora lo que nos queda es crear una conexión con una base de datos MySQL/MariaDB.

### 3.2 Conexión a través del driver JDBC

Una vez cargado el driver JDBC, con el botón de la derecha del ratón sobre el nodo MySQL (Connector/J driver), hacemos clic en la opción *Connect Using...* y aparecerá una ventana que muestra:

- El nombre del driver que estamos utilizando, en este caso *MySQL (Connector/J driver)*.
- La URL donde se encuentra la base de datos *jdbc:mysql://<HOST>:<PORT>/<DB>*, donde *<HOST>* es la máquina donde vamos a conectar, *<PORT>* es el puerto donde escucha el servidor de Bases de datos.
- *Database* es la instancia de la base de datos a la cual vamos a conectar.



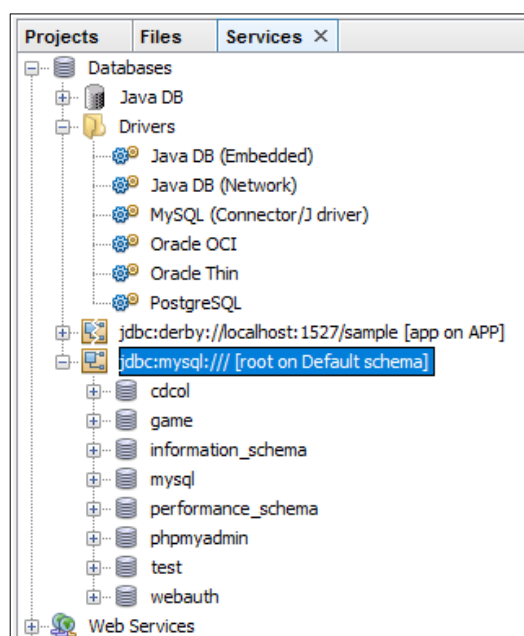
Después especificamos el usuario *root* y la contraseña de acceso a la base de datos (por defecto en blanco).

Y por último en el campo *JDBC URL:* dejamos ***`jdbc:mysql:///`*** (eliminado lo que pueda haber detrás)

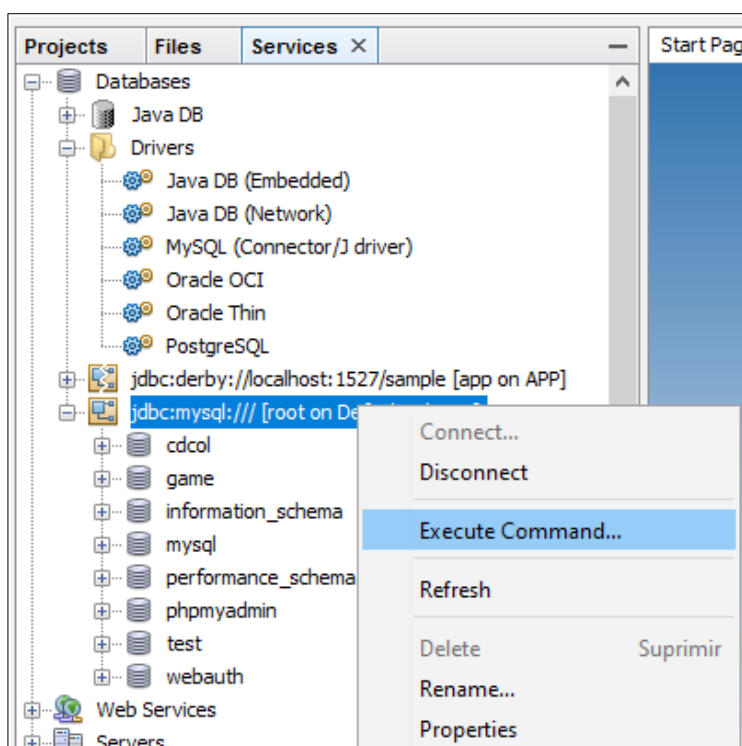
Pinchamos en *Finalizar* y si todo ha ido bien nos creará un nodo llamado ***`jdbc:mysql:///`****[root on Default schema]*

The screenshot shows the 'New Connection Wizard' dialog box, specifically the 'Customize Connection' step. The 'Driver Name' is set to 'MySQL (Connector/J driver)'. The 'Host' and 'Port' fields are empty. The 'Database' field is empty. The 'User Name' is set to 'root'. The 'Password' field is empty, and the 'Remember password' checkbox is unchecked. There are 'Connection Properties' and 'Test Connection' buttons. The 'JDBC URL' field contains 'jdbc:mysql:///'. At the bottom, there are '< Back', 'Next >', 'Finish', 'Cancel', and 'Help' buttons.

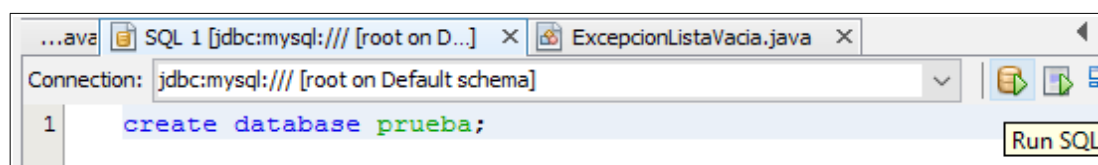
Si desplegamos el nodo `jdbc:mysql:///[root on Default schema]` veremos que aparecen el conjunto de bases de datos, tablas, vistas y procedimientos almacenados de la base de datos a la cual estamos conectada (a cada uno os aparecerán las bases de datos que tengáis en vuestro sistema). En nuestro caso todavía no hemos especificado la instancia de la base de datos a la que nos vamos a conectar ya que hemos realizado una conexión por defecto al servidor MySQL/MariaDB.

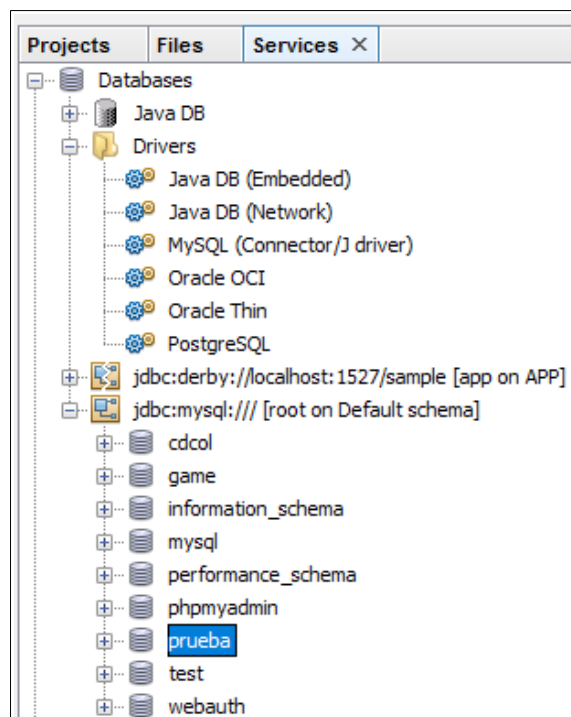


Sobre el nodo `jdbc:mysql:/// [root on Default schema]` con el botón de la derecha de ratón hacemos clic en la opción *Execute Command...* aparecerá una ventana donde podemos introducir sentencias SQL tanto de creación como de consulta sobre estructuras de la base de datos.



Nos situamos en la ventana y tecleamos lo siguiente: ***create database prueba;*** y pulsamos sobre el primer icono (*Run SQL*), si todo ha ido bien la base de datos prueba debe aparecer en el árbol. Para cada consulta SQL nos devolverá si el comando se ha realizado con éxito o por el contrario ha habido algún problema.



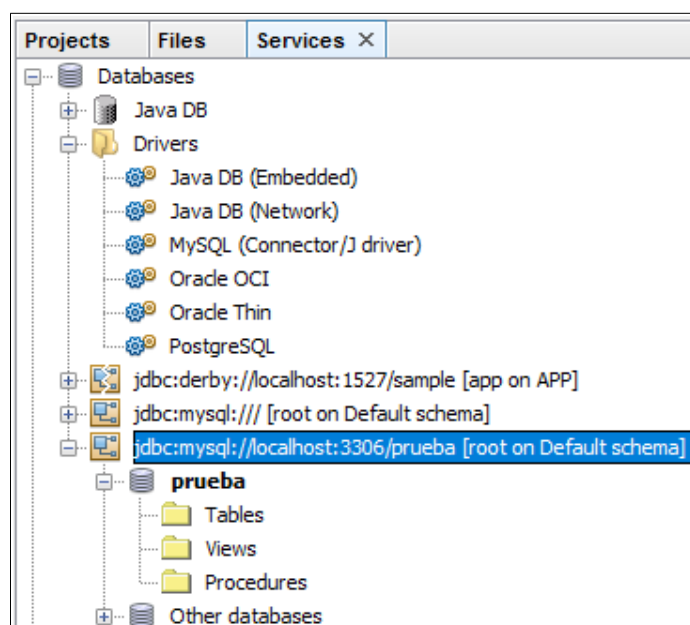
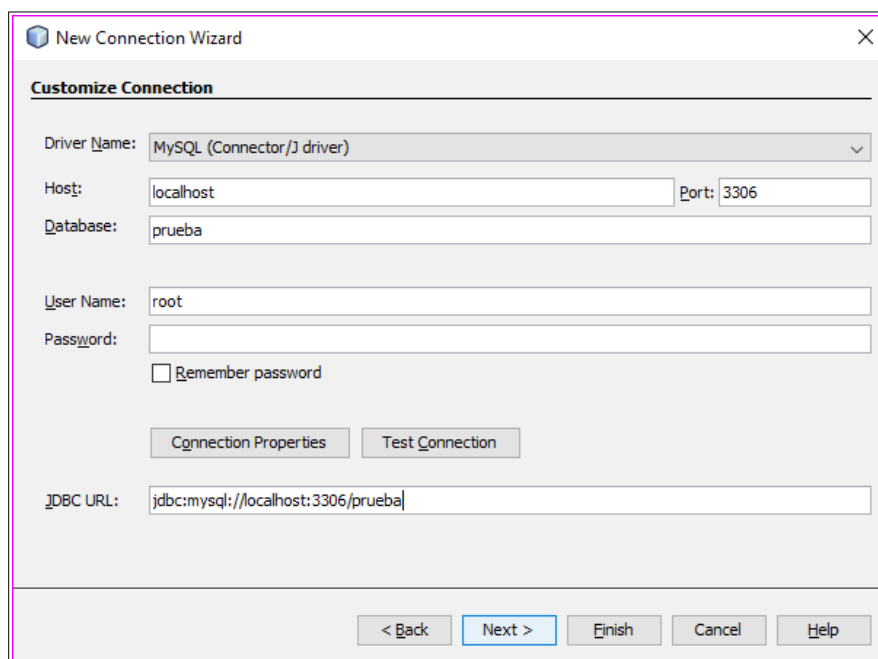


### 3.3 Conectar con la Base de Datos

Para conectar con la base de datos que hemos creado en el ejemplo anterior, volvemos a seleccionar el driver *MySQL (Connector/J driver)* y con el botón de la derecha hacemos clic y seleccionamos la opción *Connect Using...*, en este caso vamos a conectar directamente con la base de datos *prueba* que acabamos de crear, para ello escribiremos lo siguiente en el campo *Database URL*: `jdbc:mysql://localhost:3306/prueba`.

Introducimos el usuario *root* y la contraseña de acceso (por defecto en blanco), hacemos clic en el botón *Finalizar* y nos debe de crear una conexión con la instancia *prueba*.





### 3.4 Crear una tabla

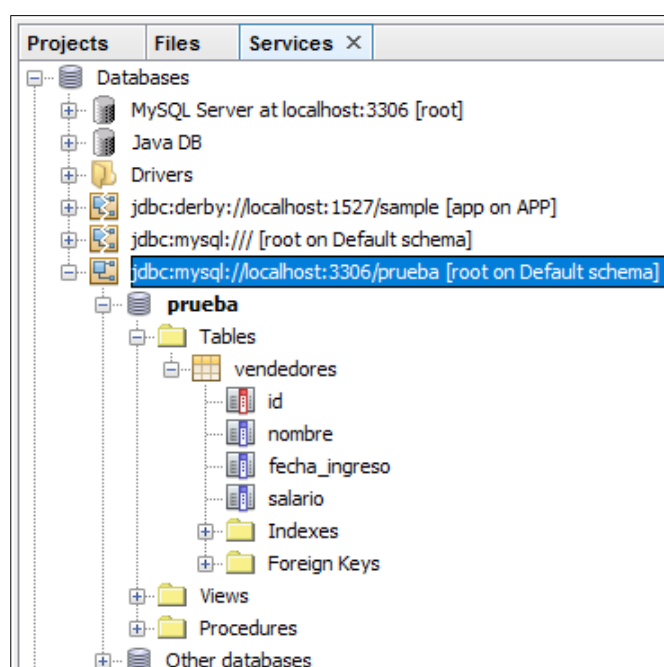
Desde el IDE de NetBeans podemos crear estructuras de dos formas, bien por sentencias SQL de creación o a partir de un asistente de creación de tablas.

#### 3.4.1 A través de sentencias SQL

Seleccionamos el nodo `jdbc:mysql://localhost:3306/prueba [root on Default schema]` y con el botón de la derecha del ratón accedemos a *Execute Command...* Introducimos el siguiente comando SQL:

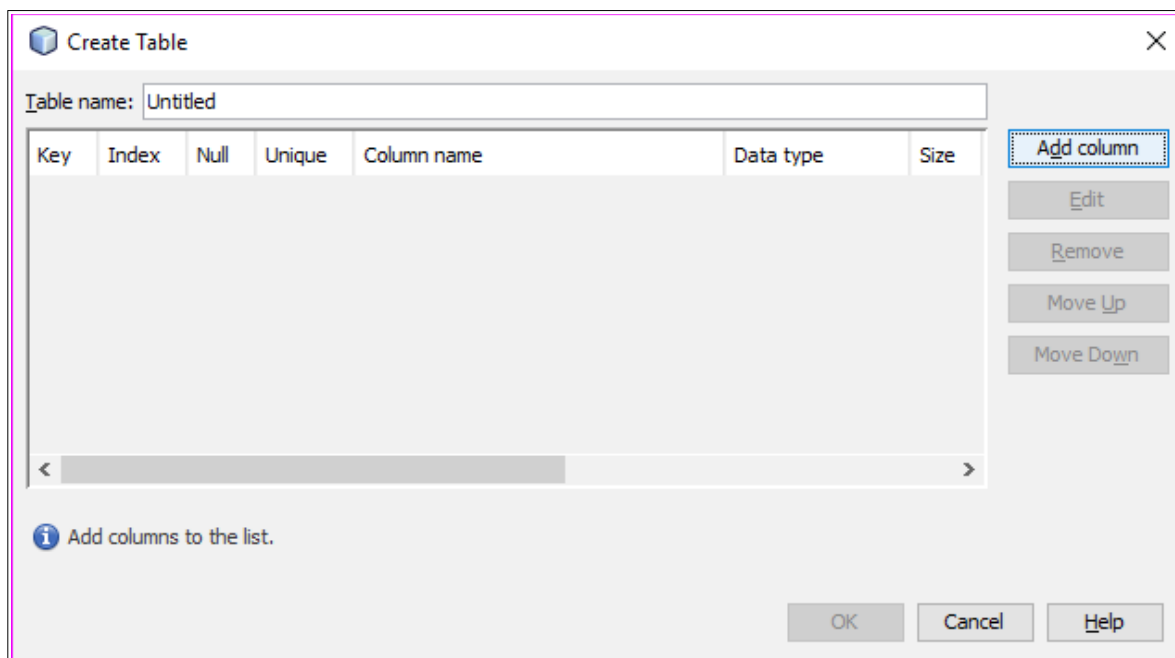
```
CREATE TABLE `vendedores` (  
  `id` int NOT NULL auto_increment,  
  `nombre` varchar(50) NOT NULL default '',  
  `fecha_ingreso` date NOT NULL default '0000-00-00',  
  `salario` float NOT NULL default '0',  
  PRIMARY KEY (`id`));
```

Si todo ha ido bien, seleccionamos el nodo de *Tables* botón derecha ratón *Refresh* y nos debe aparecer la nueva tabla que hemos creado.

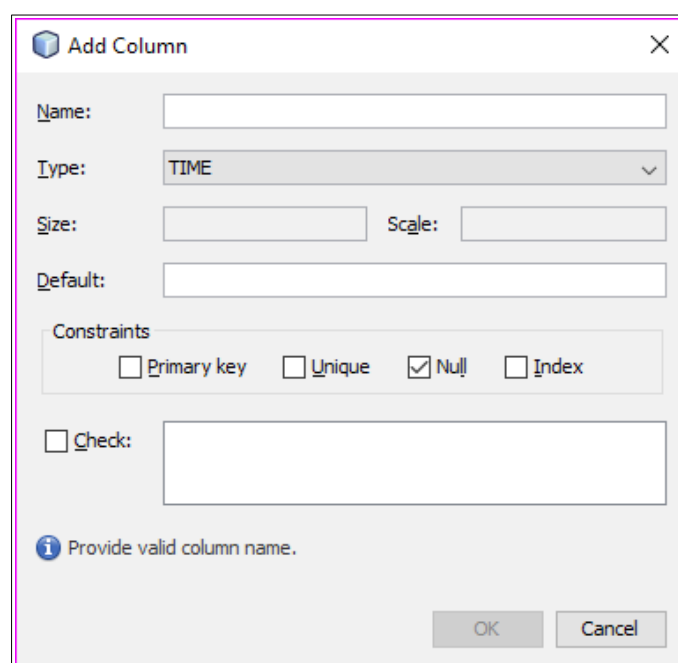


### 3.4.2 A través del asistente de creación de tablas

Desplegamos el nodo *jdbc:mysql://localhost:3306/prueba [root on Default schema]*, y seleccionamos el nodo *Tables*, con el botón de la derecha del ratón hacemos clic en la opción *Create Table*.



Pinchamos sobre *Add column* y rellenamos los datos de cada columna:



### 3.5 Insertar datos en una tabla

*Para insertar datos en una tabla lo hacemos mediante el uso de comandos SQL a través del IDE.*

*Sobre el nodo jdbc:mysql://localhost:3306/prueba con el botón derecho del ratón hacemos clic en la opción Execute Command... E introducimos el siguiente comando SQL:*

```
INSERT INTO `vendedores` VALUES (1, 'Pedro Gil', '2017-04-11', 15000);
```

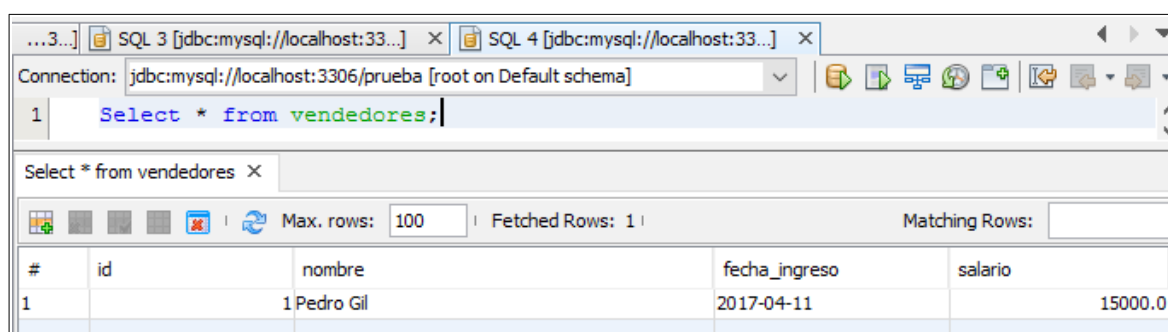
### 3.6 Consultar valores

Para realizar consulta de datos de una base de datos lo hacemos por medio de consultas SQL.

En nuestro caso vamos a ver los valores que contiene la tabla `vendedores`. Introducimos el siguiente comando SQL: (El `*` indica todos los campos)

```
Select * from vendedores;
```

Siendo el resultado:



The screenshot shows a SQL IDE window with a connection to 'jdbc:mysql://localhost:3306/prueba [root on Default schema]'. The query 'Select \* from vendedores;' is entered in the SQL editor. Below the editor, the results are displayed in a table with 1 row and 5 columns: #, id, nombre, fecha\_ingreso, and salario.

#	id	nombre	fecha_ingreso	salario
1		1 Pedro Gil	2017-04-11	15000.0

## 4. CONEXIÓN A UNA BASE DE DATOS

La API JDBC se proporciona en dos paquetes: *java.sql* y *javax.sql*. El primer paquete, que forma parte de J2SE, contiene las interfaces y clases fundamentalmente de JDBC. De estas clases, cualquier aplicación java utilizará casi siempre cuatro de ellas y en el orden especificado:

- La clase **DriverManager**: La clase *java.sql.DriverManager* es el nivel o capa gestora del API JDBC, trabaja entre el usuario y los drivers. Tiene en cuenta los drivers disponibles y a partir de ellos establece una conexión entre una base de datos y el driver adecuado para esa base de datos. Además de esta labor principal, el *DriverManager* se ocupa de mostrar mensajes de log (registro de actividad) del driver y también el tiempo límite en espera de conexión del driver, es decir, el *time-out*. Normalmente, en un programa Java el único método que un programador deberá utilizar de la clase *DriverManager* es el método **getConnection()**. Como su nombre indica este método establece una conexión con una base de datos.
- **Connection**: Una vez que las clases de los drivers se han cargado y registrado con el *DriverManager*, éstas están disponibles para establecer una conexión con una base de datos.

Cuando se realiza una petición de conexión con una llamada al método *DriverManager.getConnection()*, la clase *DriverManager* chequea cada uno de los drivers disponibles para comprobar si puede establecer la conexión. Un objeto *Connection* representa una conexión con una base de datos. Una sesión de conexión con una base de

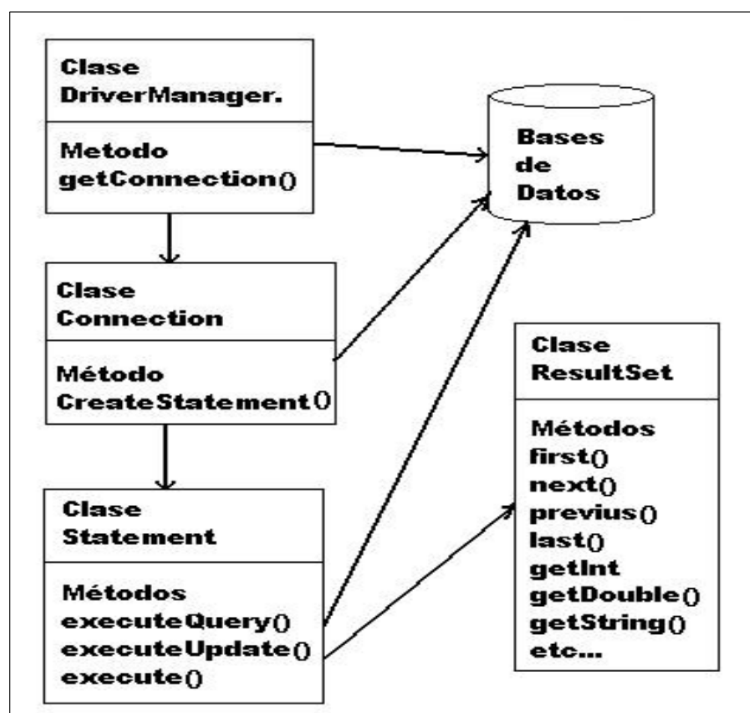
datos incluye las sentencias SQL que se ejecuten y los resultados que devuelvan.

Una sola aplicación puede tener una o más conexiones con una misma base de datos, o puede tener varias conexiones con diferentes bases de datos. *Connection* es un interfaz, ya que como se había dicho anteriormente, JDBC ofrece una plantilla o especificación que deben implementar los fabricantes de drivers de JDBC.

- **Statement:** Objeto que a través de su método *executeQuery* permite ejecutar una sentencia SQL sobre la base de datos. Devolverá un Objeto **ResultSet**.

Una vez que se ha establecido una conexión a una base de datos determinada, esta conexión se puede utilizar para enviar sentencias SQL a la base de datos. Un objeto *Statement* se crea con el método *createStatement()* de la clase *Connection*.

- **executeQuery()** Se utiliza con sentencias *SELECT* y devuelve un *ResultSet*.
  - **executeUpdate()** Se utiliza con sentencias *INSERT*, *UPDATE* y *DELETE*, o bien, con sentencias *DDL SQL*.
  - **execute()** Utilizado para cualquier sentencia *DDL*, *DML*, *DQL* o comando específico de la base de datos.
- **ResultSet:** Una tabla de datos que representa al conjunto de resultados generado al ejecutar una sentencia SQL sobre la base de datos. Los métodos de esta clase (*first()*, *next()*, *previous()*, *last()*, *getInt()*, *getDouble()*, *getString()*, etc..) nos permitirán acceder a los registros del conjunto de resultados obtenido.



Un objeto *ResultSet* es una tabla que utiliza un cursor para indicar la fila sobre la que se realizará una determinada operación. Inicialmente, este cursor está situado antes de la primera fila. Para

mover el cursor a otra fila, la interfaz *ResultSet* proporciona varios métodos. Algunos de ellos son:

- ***beforeFirst()***. Mover el cursor antes de la primera fila.
- ***first()***. Mover el cursor a la primera fila.
- ***last()***. Mover el cursor a la última fila.
- ***afterLast()***. Mover el cursor después de la última fila.
- ***previous()***. Mover el cursor a la fila anterior.
- ***next()***. Mover el cursor a la siguiente fila.

Todos los métodos devuelven un valor *true* o *false* para indicar si fue o no posible el movimiento.

Para obtener los datos de la fila donde está el cursor, la interfaz *ResultSet* también proporciona varios métodos; algunos de ellos se muestran a continuación:

- ***getString(String)***. Recupera la columna especificada. Por ejemplo:  
`cdr.getString("nombre")`
- ***getString(Int)***. Recupera la columna indicada por el índice especificado. La primera columna tiene el índice 1. Por ejemplo:  
`cdr.getString(i)`

Existen métodos análogos a los anteriores para obtener el valor de una columna de la fila actual cuando éste es entero o real. Por ejemplo:

- `getInt(String)` o `getInt(Int)`.
- `getLong(String)` o `getLong(Int)`.
- `getFloat(String)` o `getFloat(Int)`.
- `getDouble(String)` o `getDouble(Int)`.
- etc.

⚡ **Importante:** para que todo funcione, sólo falta añadir el driver a la librería del proyecto, para ello pulsaremos con el botón derecho sobre la carpeta *Libraries* y seleccionaremos *add JAR/Folder*.

## 5. EJECUCIÓN DE CONSULTAS

El método *createStatement()* del interfaz *Connection* se encuentra sobrecargado, si utilizamos su versión sin parámetros, a la hora de ejecutar sentencias SQL sobre el objeto *Statement* que se ha creado, se obtendrá el tipo de objeto *ResultSet* por defecto, es decir, se obtendría un tipo de cursor de sólo lectura y con movimiento únicamente hacia adelante.

Pero la otra versión que ofrece el interfaz *Connection* del método *createStatement()* ofrece dos parámetros que nos permiten definir el tipo de objeto *ResultSet* que se va a devolver como resultado de la ejecución de una sentencia SQL, como se muestra en el fragmento de código

siguiente:

```
String jdbcUrl = "jdbc:mysql://localhost:3306/empresa";
Connection conn = DriverManager.getConnection(jdbcUrl,"root","");
//Uso del método executeQuery
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
String sql = "select * from articulos";
rs = stmt.executeQuery(sql);
```

El primero de los parámetros indica el tipo de objeto *ResultSet* que se va a crear, y el segundo de ellos indica si el *ResultSet* es sólo de lectura o si permite modificaciones, este parámetro también se denomina tipo de concurrencia. Si especificamos el tipo de objeto *ResultSet* es obligatorio indicar si va a ser de sólo lectura o no. Si no indicamos ningún parámetro en el método *createStatement()*, se creará un objeto *ResultSet* con los valores por defecto.

Los tipos de *ResultSet* distintos que se pueden crear dependen del valor del primer parámetro, estos valores se corresponden con constantes definidas en el interfaz *ResultSet*. Estas constantes se describen a continuación:

- *TYPE\_FORWARD\_ONLY*: se crea un objeto *ResultSet* con movimiento únicamente hacia delante (*forward-only*). Es el tipo de *ResultSet* por defecto.
- *TYPE\_SCROLL\_INSENSITIVE*: se crea un objeto *ResultSet* que permite todo tipo de movimientos. Pero este tipo de *ResultSet*, mientras está abierto, no será consciente de los cambios que se realicen sobre los datos que está mostrando, y por lo tanto no mostrará estas modificaciones.
- *TYPE\_SCROLL\_SENSITIVE*: al igual que el anterior permite todo tipo de movimientos, y además permite ver los cambios que se realizan sobre los datos que contiene.

Los valores que puede tener el segundo parámetro que define la creación de un objeto *ResultSet*, son también constantes definidas en el interfaz *ResultSet* y son las siguientes:

- *CONCUR\_READ\_ONLY*: indica que el *ResultSet* es sólo de lectura. Es el valor por defecto.
- *CONCUR\_UPDATABLE*: permite realizar modificaciones sobre los datos que contiene el *ResultSet*.

En un objeto *ResultSet* se encuentran los resultados de la ejecución de una sentencia SQL, por lo tanto, un objeto *ResultSet* contiene las filas que satisfacen las condiciones de una sentencia SQL, y ofrece el acceso a los datos de las filas a través de una serie de métodos *getXXX()* que permiten acceder a las columnas de la fila actual.

El método *next()* del interfaz *ResultSet* es utilizado para desplazarse a la siguiente fila del *ResultSet*,

haciendo que la próxima fila sea la actual, además de este método de desplazamiento básico, según el tipo de *ResultSet* podremos realizar desplazamientos libres utilizando método como *last()*, *relative()* o *previous()*.

El aspecto que suele tener un *ResultSet* es una tabla con cabeceras de columnas y los valores correspondientes devueltos por una consulta. Un *ResultSet* mantiene un cursor que apunta a la fila actual de datos. El cursor se mueve hacia abajo cada vez que el método *next()* es ejecutado.

⚡ Inicialmente el cursor de *ResultSet* está posicionado antes de la primera fila, de esta forma, la primera llamada a *next()* situará el cursor en la primera fila, pasando a ser la fila actual.

Las filas del *ResultSet* son devueltas de arriba a abajo según se va desplazando el cursor con las sucesivas llamadas al método *next()*. Un cursor es válido hasta que el objeto *ResultSet* o su objeto padre *Statement* es cerrado.

### 5.1 Obteniendo datos del *ResultSet*

Los métodos *getXXX()* ofrecen los medios para recuperar los valores de las columnas (campos) de la fila (registro) actual del *ResultSet*. Dentro de cada columna no es necesario que las columnas sean recuperadas utilizando un orden determinado, pero para una mayor portabilidad entre diferentes bases de datos se recomienda que los valores de las columnas se recuperen de izquierda a derecha y solamente una vez.

Para designar una columna podemos utilizar su nombre o bien su número de orden.

Por ejemplo si la segunda columna de un objeto *rs* de la clase *ResultSet* se llama "título" y almacena datos de tipo *String*, se podrá recuperar su valor de las formas que muestra el siguiente fragmento de código:

```
//rs es un objeto de tipo ResultSet
String valor=rs.getString(2);
String valor=rs.getString("titulo");
```

⚡ Se debe señalar que las **columnas se numeran de izquierda a derecha** empezando con la columna 1, y que los **nombres** de las **columnas no son case sensitive**, es decir, no distinguen entre mayúsculas y minúsculas.

📖 La información referente a las columnas que contiene el *ResultSet* se encuentra disponible llamando al método *getMetaData()*, este método devolverá un objeto *ResultSetMetaData* que contendrá el número, tipo y propiedades de las columnas del *ResultSet*.




Si conocemos el nombre de una columna, pero no su índice, el método *findColumn()* puede ser utilizado para obtener el número de columna, pasándole como argumento un objeto *String* que sea el nombre de la columna correspondiente, este método nos devolverá un entero que será el índice correspondiente a la columna.

## 5.2 Tipos de datos y conversiones

Cuando se lanza un método *getXXX()* determinado sobre un objeto *ResultSet* para obtener el valor de un campo del registro actual, el driver JDBC convierte el dato que se quiere recuperar al tipo Java especificado y entonces devuelve un valor Java adecuado. Por ejemplo si utilizamos el método *getString()* y el tipo del dato en la base de datos es *VARCHAR*, el driver JDBC convertirá el dato *VARCHAR* a un objeto *String* de Java, por lo tanto el valor de retorno de *getString()* será un objeto de la clase *String*.

## 5.3 Desplazamiento en un *ResultSet*

El método *next()* del interfaz *ResultSet* lo vamos a utilizar para desplazarnos al registro siguiente dentro de un *ResultSet*.

 El método *next()* devuelve un valor booleano (tipo *boolean* de Java), *true* si el registro siguiente existe y *false* si hemos llegado al final del objeto *ResultSet*, es decir, no hay más registros.

Además de tener el método *next()*, disponemos de los siguientes métodos para el desplazamiento y movimiento dentro de un objeto *ResultSet*:

- ***boolean absolute(int registro)***: desplaza el cursor al número de registros indicado. Si el valor es negativo, se posiciona en el número registro indicado pero empezando por el final. Este método devolverá *false* si nos hemos desplazado después del último registro o antes del primer registro del objeto *ResultSet*. Para poder utilizar este método el objeto *ResultSet* debe ser de tipo *TYPE\_SCROLL\_SENSITIVE* o de tipo *TYPE\_SCROLL\_INSENSITIVE*, a un *ResultSet* que es de cualquiera de estos dos tipos se dice que es de tipo *scrollable*. Si a este método le pasamos un valor cero se lanzará una excepción *SQLException*.
- ***void afterLast()***: se desplaza al final del objeto *ResultSet*, después del último registro. Si el *ResultSet* no posee registros, este método no tienen ningún efecto. Este método sólo se puede utilizar en objetos *ResultSet* de tipo *scrollable*.
- ***void beforeFirst()***: mueve el cursor al comienzo del objeto *ResultSet*, antes del primer registro. Sólo se puede utilizar sobre objetos *ResultSet* de tipo *scrollable*.

- ***boolean first()***: desplaza el cursor al primer registro. Devuelve *true* si el cursor se ha desplazado a un registro válido, por el contrario, devolverá *false* en otro caso o bien si el objeto *ResultSet* no contiene registros. Al igual que los métodos anteriores, sólo se puede utilizar en objetos *ResultSet* de tipo *scrollable*.
- ***void last()***: desplaza el cursor al último registro del objeto *ResultSet*. Devolverá *true* si el cursor se encuentra en un registro válido, y *false* en otro caso o si el objeto *ResultSet* no tiene registros. Sólo es válido para objetos *ResultSet* de tipo *scrollable*, en caso contrario lanzará una excepción *SQLException*.
- ***void moveToCurrentRow()***: mueve el cursor a la posición recordada, normalmente el registro actual. Este método sólo tiene sentido cuando estamos situados dentro del *ResultSet* en un registro que se ha insertado. Este método sólo es válido utilizarlo con objetos *ResultSet* que permiten la modificación, es decir, están definidos mediante la constante *CONCUR\_UPDATABLE*.
- ***boolean previous()***: desplaza el cursor al registro anterior. Es el método contrario al método *next()*. Devolverá *true* si el cursor se encuentra en un registro o fila válidos, y *false* en caso contrario. Sólo es válido este método con objetos *ResultSet* de tipo *scrollable*, en caso contrario lanzará una excepción *SQLException*.
- ***boolean relative(int registros)***: mueve el cursor un número relativo de registros, este número puede ser positivo o negativo. Si el número es negativo el cursor se desplazará hacia el principio del objeto *ResultSet* el número de registros indicados, y si es positivo se desplazará hacia el final de objeto *ResultSet* correspondiente. Este método sólo se puede utilizar si el *ResultSet* es de tipo *scrollable*. También existen otros métodos dentro del interfaz *ResultSet* que están relacionados con el desplazamiento:
  - ***boolean isAfterLast()***: indica si nos encontramos después del último registro del objeto *ResultSet*. Sólo se puede utilizar en objetos *ResultSet* de tipo *scrollable*.
  - ***boolean isBeforeFirst()***: indica si nos encontramos antes del primer registro del objeto *ResultSet*. Sólo se puede utilizar en objetos *ResultSet* de tipo *scrollable*.
  - ***boolean isFirst()***: indica si el cursor se encuentra en el primer registro. Sólo se puede utilizar en objetos *ResultSet* de tipo *scrollable*.
  - ***boolean isLast()***: indica si nos encontramos en el último registro del *ResultSet*. Sólo se puede utilizar en objetos *ResultSet* de tipo *scrollable*.
- ***int getRow()***: devuelve el número de registro actual. El primer registro será el número 1, el segundo el 2, etc. Devolverá cero si no hay registro actual.

## 6. MODIFICACIÓN

Para poder modificar los datos que contiene un *ResultSet* debemos crear un *ResultSet* de tipo modificable, para ello debemos utilizar la constante *ResultSet.CONCUR\_UPDATABLE* dentro del método *createStatement()*.

Para modificar los valores de un registro existente se utilizan una serie de métodos *updateXXX()* del interfaz *ResultSet*. Las XXX indican el tipo del dato al igual que ocurre con los métodos *getXXX()* de este mismo interfaz. El proceso para realizar la modificación de una fila de un *ResultSet* es el siguiente:

1. Nos situamos sobre el registro que queremos modificar.
2. Lanzamos los métodos *updateXXX()* adecuados, pasándole como argumento los nuevos valores.
3. A continuación lanzamos el método *updateRow()* para que los cambios tengan efecto sobre la base de datos.

El método *updateXXX()* recibe dos parámetros, el campo o columna a modificar y el nuevo valor. La columna la podemos indicar por su número de orden o bien por su nombre, igual que en los métodos *getXXX()*.

El siguiente fragmento de código muestra cómo se puede modificar el campo *dirección* del último registro de un *ResultSet* que contiene el resultado de una *SELECT* sobre la tabla de *clientes*:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
//Ejecutamos la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
System.out.println("Situamos el cursor al final");
//Nos situamos en el último registro del ResultSet y hacemos la modificación
rs.last();
rs.updateString("direccion", "C/ Pepe Ciges, 3");
rs.updateRow();
```

Si nos desplazamos dentro del *ResultSet* antes de lanzar el método *updateRow()*, se perderán las modificaciones realizadas. Y si queremos cancelar las modificaciones lanzaremos el método *cancelRowUpdates()* sobre el objeto *ResultSet*, en lugar del método *updateRow()*.

Una vez que hemos invocado el método *updateRow()*, el método *cancelRowUpdates()* no tendrá ningún efecto. El método *cancelRowUpdates()* cancela las modificaciones de todos los campos de un registro, es decir, si hemos modificado dos campos con el método *updateXXX()* se cancelarán ambas modificaciones.

## 7. INSERCIÓN Y BORRADO

Además de poder realizar modificaciones directamente sobre las filas de un *ResultSet*, también podemos añadir nuevas filas (registros) y eliminar las existentes.

Estos métodos son:

- *insertRow()*
- *moveToInsertRow()*
- *deleteRow()*

El primer paso para insertar un registro o fila en un *ResultSet* es mover el cursor (puntero que indica el registro actual) del *ResultSet*, esto se consigue mediante el método *moveToInsertRow()*.

El siguiente paso es dar un valor a cada uno de los campos que van a formar parte del nuevo registro, para ello se utilizan los métodos *updateXXX()* adecuados.

Para finalizar el proceso se lanza el método *insertRow()*, que creará el nuevo registro tanto en el *ResultSet* como en la tabla de la base de datos correspondientes.

Hasta que no se lanza el método *insertRow()*, la fila no se incluye dentro del *ResultSet*, es una fila especial denominada "fila de inserción" (*insert row*) y es similar a un *buffer* completamente independiente del objeto *ResultSet*.

El siguiente fragmento de código da de alta un nuevo registro en la tabla de *clientes*:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
//Ejecutamos la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
//Creamos un nuevo registro en la tabla de clientes
rs.moveToInsertRow();
rs.updateString(2, "Killy Lopez");
rs.updateString(3, "Wall Street 3674");
rs.insertRow();
```

Si no facilitamos valores a todos los campos del nuevo registro con los métodos *updateXXX()*, ese campo tendrá un valor *NULL*, y si en la base de datos no está definido ese campo para admitir nulos se producirá una excepción *SQLException*.

Cuando hemos insertado nuestro nuevo registro en el objeto *ResultSet*, podremos volver a la antigua posición en la que nos encontrábamos dentro del *ResultSet*, antes de haber lanzado el método *moveToInsertRow()*, llamando al método *moveToCurrentRow()*, este método sólo se puede utilizar en combinación con el método *moveToInsertRow()*.

Además de insertar filas en nuestro objeto *ResultSet* también podremos eliminar filas o registros del mismo. El método que se debe utilizar para esta tarea es el método *deleteRow()*. Para eliminar un registro no tenemos que hacer nada más que movernos a ese registro, y lanzar el método *deleteRow()* sobre el objeto *ResultSet* correspondiente.

El siguiente fragmento de código borra el último registro de la tabla de clientes:

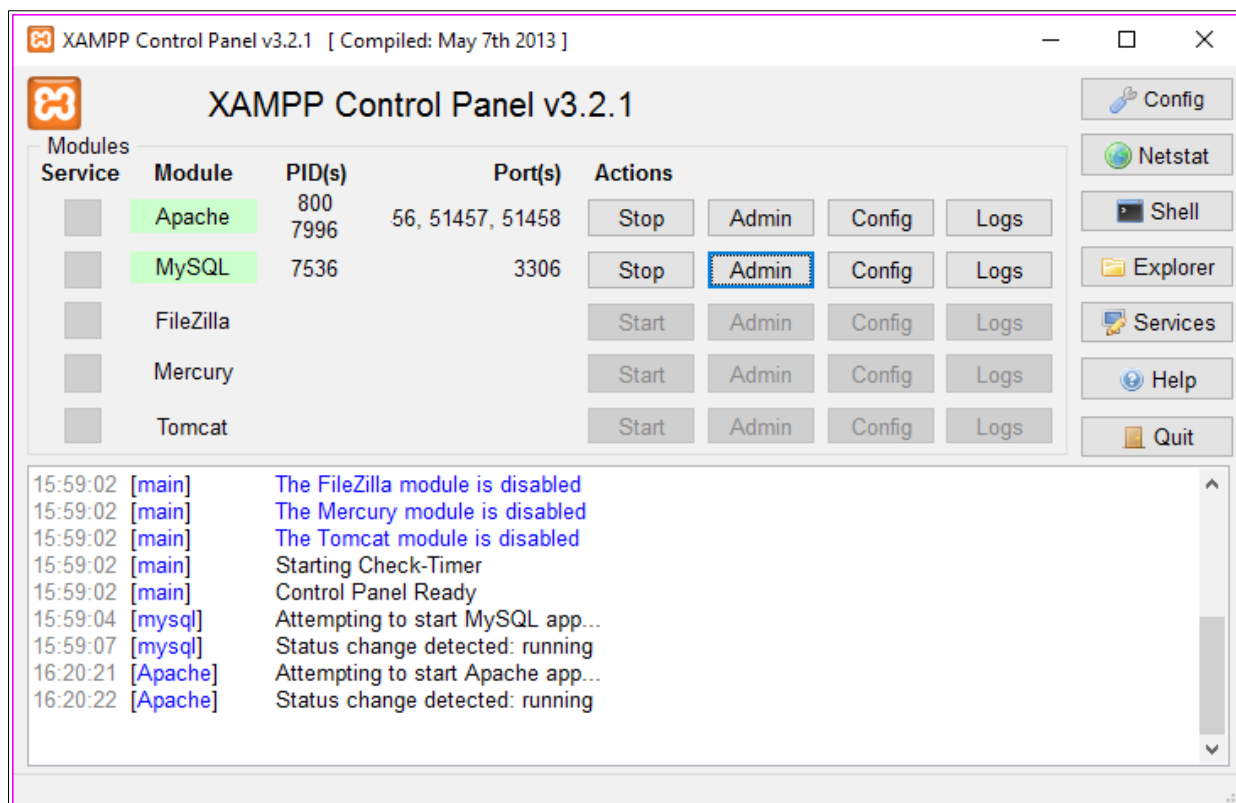
```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
//Ejecutamos la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
//Nos situamos al final del ResultSet
rs.last();
rs.deleteRow();
```

## 8. EJEMPLO

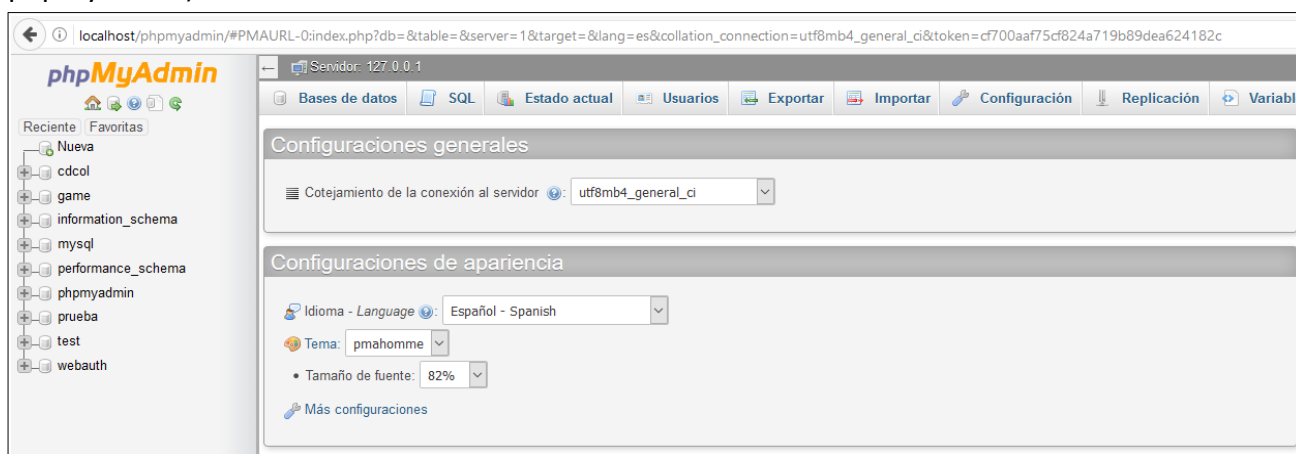
Vamos a ver un ejemplo completo de cómo trabajar con bases de datos en Java. El ejemplo se basa en una aplicación donde vamos a dar de alta, de baja, modificar y consultar todos los clientes de una tienda.

En la base de datos tendremos una sola tabla con los campos *id*, *nombre* y *dirección*. Cabe destacar que el campo *id* es autoincremental, es decir que su valor se incrementará al insertar un registro en la tabla, por lo que no hace falta que le demos nosotros el valor.

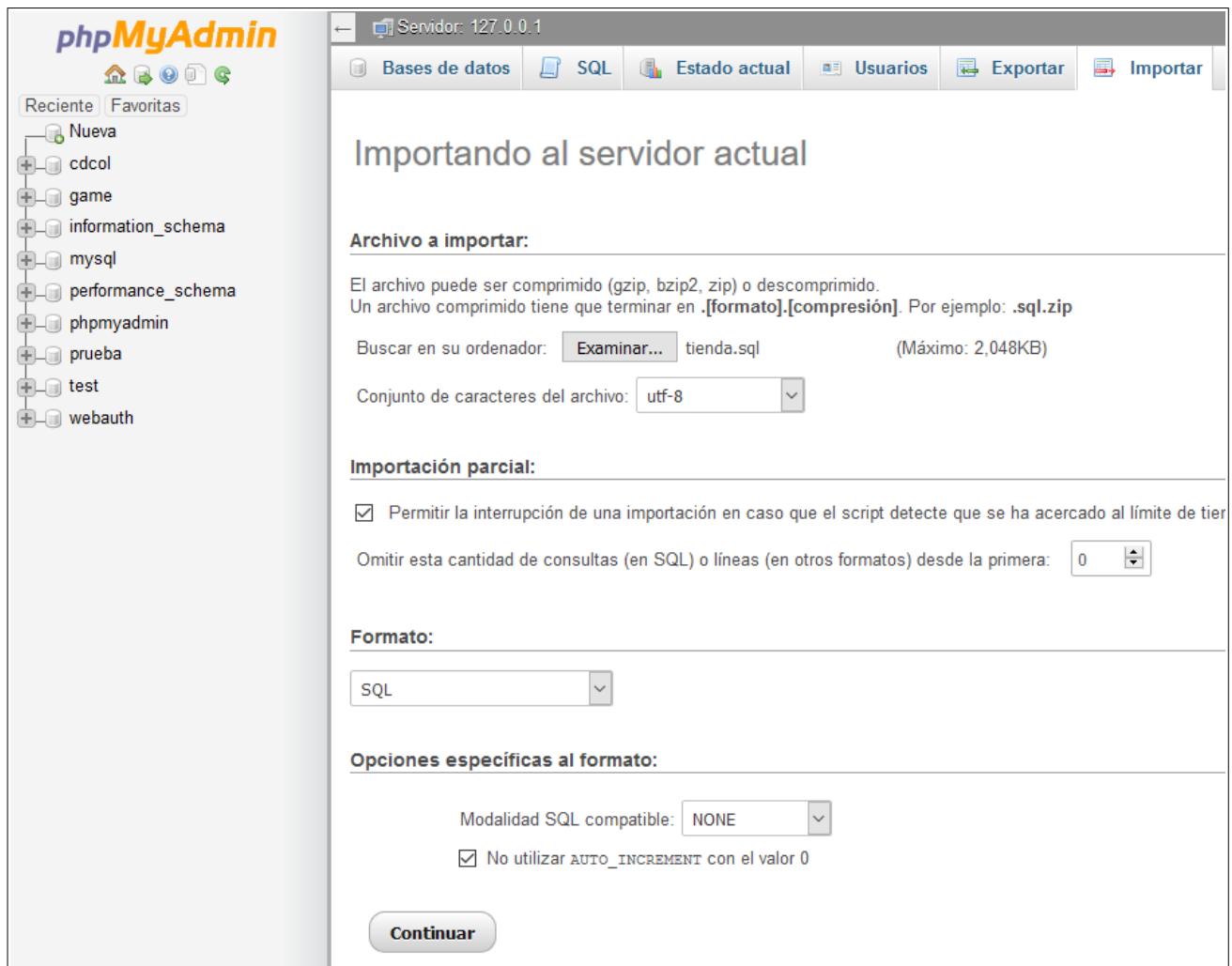
Antes de nada vamos a importar la base de datos a nuestro servidor. Para ello descargamos la base de datos *tienda.sql* del aula virtual, abrimos *XAMPP*, arrancamos *Apache* y *MySQL/MariaDB* y pinchamos en *Admin* (en la fila de *MySQL*):



Automáticamente nos abrirá en el navegador la página de administrador de bases de datos: (los que ya teníais instalado XAMPP del módulo de BD podéis abrir directamente la aplicación phpMyAdmin)

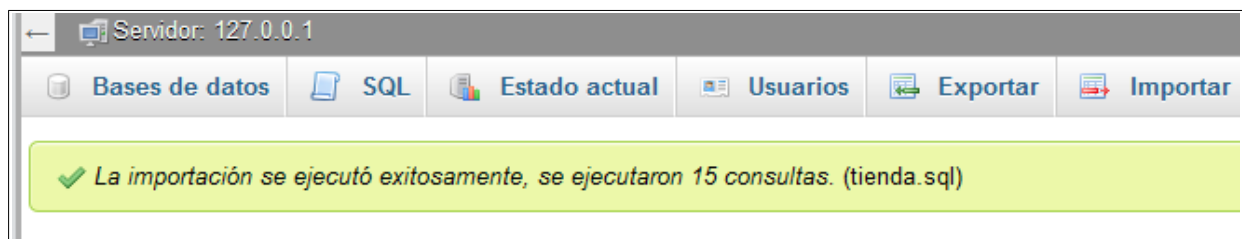


A continuación, pinchamos en Importar y seleccionamos la base de datos *tienda.sql* descargada:



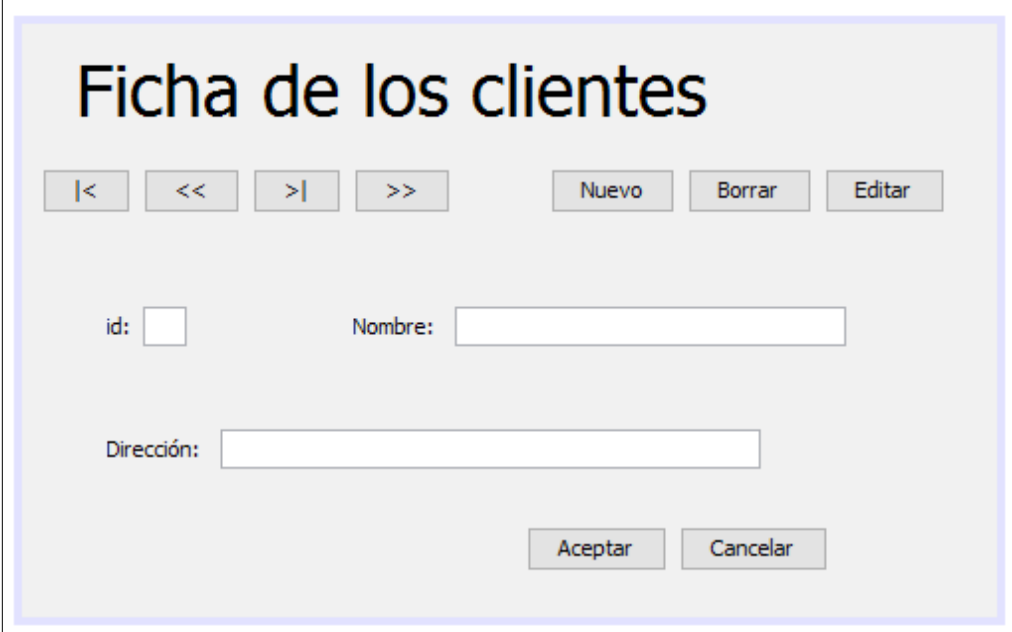
The screenshot shows the phpMyAdmin interface for importing a file. The left sidebar lists databases: Nueva, cdcol, game, information\_schema, mysql, performance\_schema, phpmyadmin, prueba, test, and webauth. The main area is titled 'Importando al servidor actual'. It includes a section 'Archivo a importar:' with instructions on file formats and a search box containing 'tienda.sql'. Below this is a dropdown for 'Conjunto de caracteres del archivo' set to 'utf-8'. The 'Importación parcial:' section has a checked checkbox for interrupting the process and a spinner for 'Omitir esta cantidad de consultas' set to 0. The 'Formato:' dropdown is set to 'SQL'. Under 'Opciones específicas al formato:', the 'Modalidad SQL compatible' dropdown is set to 'NONE' and the checkbox 'No utilizar AUTO\_INCREMENT con el valor 0' is checked. A 'Continuar' button is at the bottom.

Pinchamos en *Continuar* y si todo va bien se importará sin problemas:



Ya tenemos la base de datos en nuestro servidor. Es importante que no paremos los módulos de XAMPP para poder acceder a la base de datos.

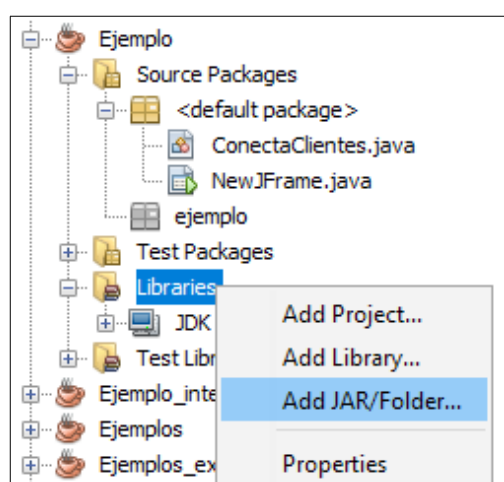
A continuación, creamos un *JFrame* y diseñamos la aplicación:



La conexión a la base de datos, el cierre, las modificaciones, etc... se realizarán en una clase diferente al *JFrame*, esta clase será la encargada de los datos de la base de datos, la llamaremos *ConectaClientes*.

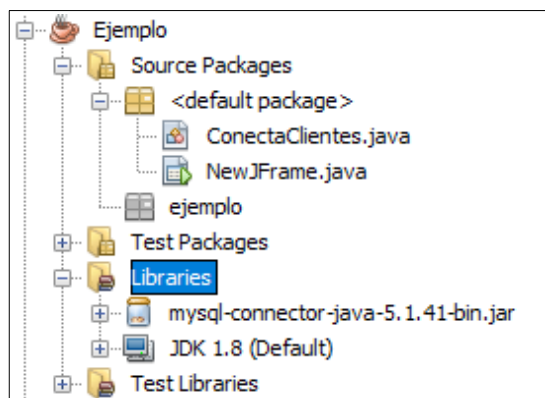
En el aula virtual tenéis el proyecto comentado.

Para poder utilizar el *Driver* de *MySQL/MariaDB* lo debemos de añadir al proyecto. Para ello, pulsamos con el botón derecho sobre *Libraries*:



Y buscamos en nuestro equipo e insertamos el *Driver mysql-connector-java-5.1.41-bin.jar*.





Es importante tener en cuenta que si no encuentra el `.jar` en la ruta proporcionada nos dará un error. Esto pasará, por ejemplo, cuando descarguéis el proyecto y no esté ubicado en la misma ruta que en el ejemplo. Tan solo tendremos que buscarlo e insertarlo de nuevo.

## 9. AGRADECIMIENTOS

Apuntes actualizados y adaptados al CEEDCV a partir de la siguiente documentación:

[1] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.