

## UNIDAD 13

### ACCESO A BASES DE DATOS

Programación  
CFGS DAW

Autores:  
Carlos Cacho y Raquel Torres

Revisado por:  
Lionel Tarazón – [lionel.tarazon@ceedcv.es](mailto:lionel.tarazon@ceedcv.es)

2019/2020


#### 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. INTRODUCCIÓN.....</b>	<b>4</b>
<b>2. REPASO DEL LENGUAJE SQL.....</b>	<b>5</b>
2.1 Comandos.....	5
2.2 Clausulas.....	6
2.3 Operadores.....	6
2.4 Ejemplos de sentencias SQL.....	6
<b>3. JDBC.....</b>	<b>9</b>
3.1 Funciones del JDBC.....	9
3.2 Drivers JDBC.....	10
<b>4. ACCESO A BASES DE DATOS DESDE NETBEANS.....</b>	<b>11</b>
4.1 Instalación del JDBC Driver.....	11
4.2 Conexión a Base de Datos mediante el JDBC Driver.....	13
4.3 Crear una tabla.....	16
4.3.1 Mediante sentencias SQL.....	16
4.3.2 Mediante el asistente de creación de tablas.....	16
4.4 Insertar datos.....	17
4.5 Consultar valores.....	17
<b>5. ACCESO A BASES DE DATOS MEDIANTE CÓDIGO JAVA.....</b>	<b>18</b>
5.1 Añadir la librería JDBC al proyecto.....	18
5.2 Cargar el Driver.....	18
5.3 Clase DriverManager.....	19
5.4 Clase Connection.....	20
5.5 Clase Statement.....	20
5.6 Clase ResultSet.....	21
5.7 Ejemplo completo.....	22
<b>6. NAVEGABILIDAD Y CONCURRENCIA.....</b>	<b>23</b>
<b>7. CONSULTAS.....</b>	<b>24</b>
7.1 Navegación de un ResultSet.....	24
7.2 Obteniendo datos del ResultSet.....	25
7.3 Tipos de datos y conversiones.....	25
<b>8. MODIFICACIÓN.....</b>	<b>26</b>
<b>9. INSERCIÓN.....</b>	<b>28</b>
<b>10. BORRADO.....</b>	<b>29</b>
<b>11. PROGRAMA GESTOR DE CLIENTES.....</b>	<b>30</b>
<b>12. AGRADECIMIENTOS.....</b>	<b>33</b>

## UD13. ACCESO A BASES DE DATOS

### 1. INTRODUCCIÓN

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 de gestión de bases de datos (SGBD); por ejemplo, Access, SQL Server, Oracle y DB2. Otros SGBD 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**.

Antes de empezar **necesitamos un sistema de gestión de base de datos** instalado en nuestro ordenador en el que poder tener bases de datos a las que conectarnos desde nuestros programas escritos en lenguaje Java. Si ya tiene uno instalado (por ejemplo MySQL, MariaDB, PostgreSQL, etc.) puedes utilizarlo si quieres. De todos modos en esta unidad os proponemos utilizar **XAMPP (Windows/Linux Apache MariaDB PHP Perl)** que incorpora un servidor web y la herramienta phpMyAdmin para trabajar con bases de datos. Lo puedes descargar desde la dirección: <https://www.apachefriends.org> Para su instalación tan solo debes seguir los pasos del instalador.



Veréis que las versiones más actuales de XAMPP han sustituido MySQL por MariaDB. No pasa nada, ambos son compatibles y pueden utilizarse indistintamente.

## 2. REPASO DEL LENGUAJE SQL

En este apartado se da un repaso a los aspectos más relevantes del lenguaje SQL de manipulación de bases de datos relacionales que ya habrás visto en el módulo de Bases de Datos. Te vendrá bien para repasar conceptos. Si no has cursado dicho módulo, es muy importante que leas este apartado con atención. No es necesario tener un conocimiento avanzado sobre el lenguaje SQL pero sí los aspectos más básicos que se cubren en este apartado.

SQL incluye sentencias tanto de creación de datos (CREATE) como de manipulación de datos (INSERT, UPDATE y DELETE) así como de consulta de datos (SELECT).

La sintaxis del lenguaje SQL está formado por cuatro grupos sintácticos que pueden combinarse en varias de las sentencias más habituales. Estos 4 grupos son: **Comandos, Cláusulas, Operadores y Funciones**. En este apartado repasaremos solo los tres primeros.

### 2.1 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.

## 2.2 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 un criterio dado.

## 2.3 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

## 2.4 Ejemplos de sentencias SQL

Para **crear una base de datos:** *CREATE DATABASE <nombre>*

Ejemplo: *CREATE DATABASE tiendaonline*

Para **eliminar una base de datos:** *DROP DATABASE <base de datos>*

Ejemplo: *DROP DATABASE tiendaonline*

Para **utilizar una base de datos:** *USE <base de datos>*

Ejemplo: *USE tiendaonline*

NOTA: Con *USE* indicamos sobre qué base de datos queremos trabajar (podemos tener varias).

Para **crear una tabla**: `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`.

Ejemplo:

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

Para **borrar una tabla**:

```
DROP TABLE <tabla>
```

Ejemplo:

```
DROP TABLE clientes
```

Para **insertar registros en una tabla**:

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

Ejemplo: Insertamos dos nuevos registros en la tabla clientes

```
INSERT INTO clientes VALUES ('Pepito Pérez','VALENCIA','963003030','Ninguna'),  
('María Martínez','VALENCIA','961002030','Ninguna')
```

Para **modificar registros ya existentes en una tabla**:

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

Ejemplo: Cambiamos el campo 'dirección' del registro cuyo teléfono es '963003030'

```
UPDATE clientes SET direccion = 'Puerto de Sagunto'  
WHERE telefono = '963003030'
```

Para **borrar registros de una tabla**:

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

Ejemplo: Borramos los registros cuyo campo 'telefono' es 963003030.

```
DELETE FROM clientes WHERE telefono='963003030'
```

Para **realizar una consulta**, es decir, obtener registros de una base de datos:

```
SELECT [ALL | DISTINCT] <lista de selección>
```

```
FROM <tablas>
```

```
WHERE <condiciones de selección>
```

```
[ORDER BY <columna1> [ASC|DESC][, <columna2>[ASC|DESC]]...]
```

Algunos ejemplos sencillos:

Obtenemos todos los registros de la tabla 'clientes':

```
SELECT * FROM clientes;
```

Obtenemos solo el campo 'nombre' de todos los registros de la tabla 'clientes':

```
SELECT nombre FROM clientes;
```

Obtenemos solo los campos 'nombre' y 'telefono' de la tabla 'clientes':

```
SELECT nombre FROM clientes;
```

Obtenemos todos los registros de 'clientes' ordenados por nombre:

```
SELECT * FROM clientes ORDER BY nombre;
```

Obtenemos solo los nombres y teléfonos de los clientes, ordenados por nombre:

```
SELECT nombre, telefono FROM clientes ORDER BY nombre;
```

Obtenemos todos los registros de 'clientes' donde el campo 'telefono' es mayor que 1234

```
SELECT * FROM clientes WHERE telefono > '1234';
```

Obtenemos todos los registros de 'clientes' en los que el campo 'telefono' empieza por 91


```
SELECT * FROM clientes WHERE telefono LIKE '91*';
```

Para ver ejemplos más complejos y/o que impliquen datos de varias tablas podéis consultar este material complementario: <https://www.cs.us.es/blogs/bd2013/files/2013/09/Consultas-SQL.pdf>




### 3. JDBC

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*.

#### 3.1 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 datos.
- Procesar los resultados de la ejecución de las sentencias.

### 3.2 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<sup>1</sup> 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.

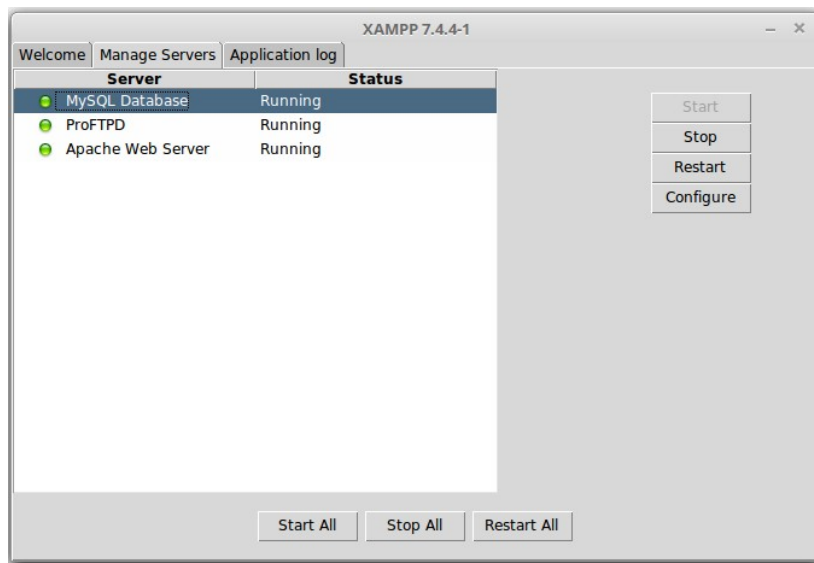
Al ser usuario de ODBC depende de las dll de ODBC y eso limita la cantidad de plataformas en donde se puede ejecutar la aplicación.

- 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.

1 ODBC = [Open DataBase Connectivity](#). Estándar de acceso a bases de datos.

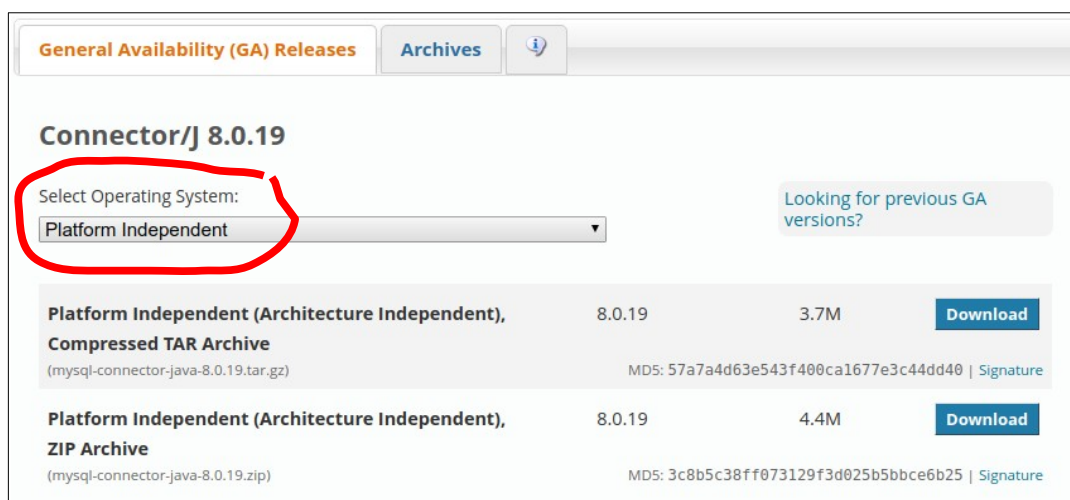
## 4. ACCESO A BASES DE DATOS DESDE NETBEANS

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



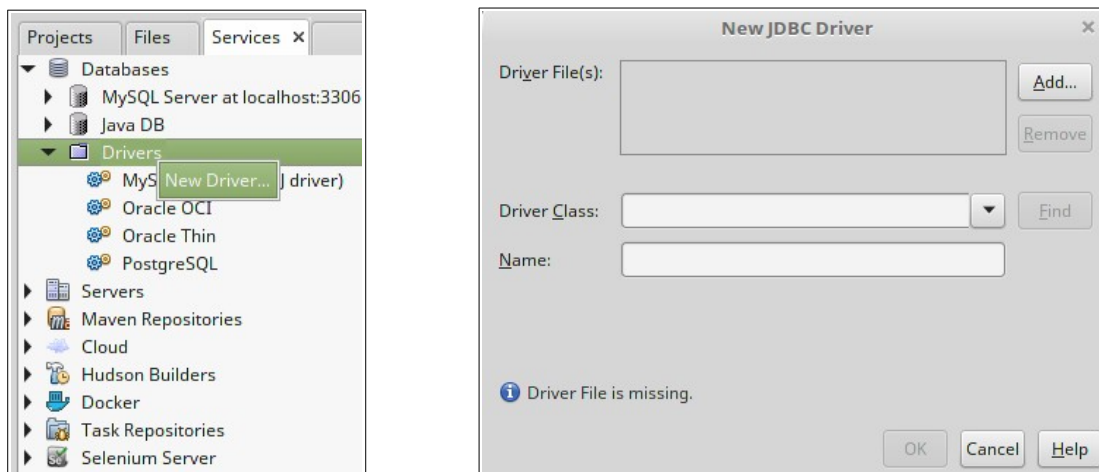
### 4.1 Instalación del JDBC Driver

Para poder conectarse a una base de datos MySQL/MariaDB desde el explorador de NetBeans necesitamos instalar el driver mysql-connector de Java. Primero hay que descargarlo de la dirección web <http://dev.mysql.com/downloads/connector/j/>. En la opción de sistema operativo debemos seleccionar "Platform Independent" y luego descargar el ZIP.

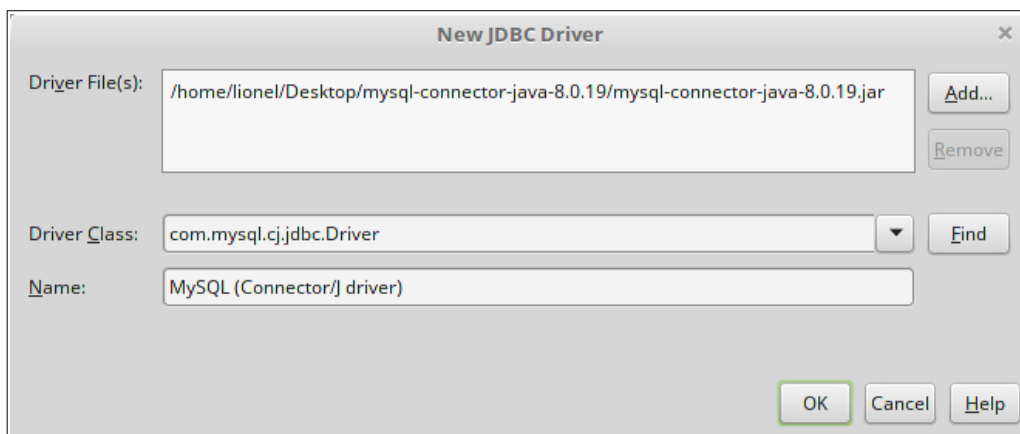


No hace falta registrarse para descargarlo, solo hay pulsar en "No thanks, just start my download." Descomprime el ZIP y busca el archivo **mysql-connector-java-8.0.19.jar**. Ten en cuenta que el número de versión puede ser diferente a la que se muestra en la imagen.

Ahora **abre NetBeans**, ve al **panel Services → Databases**, haz **clic derecho en la opción Drivers** y selecciona la opción **New Driver** del menú contextual. Se mostrará el diálogo **New JDBC Driver**.



Haz **clic en Add...** y selecciona el fichero `mysql-connector-java-8.0.19.jar` que has descargado anteriormente. Una vez seleccionado el fichero nos debe de rellenar la clase principal del Driver que en nuestro caso debe de ser `com.mysql.cj.jdbc.Driver` y el nombre a nuestro driver, por ejemplo **MySQL (Connector/J driver)**.

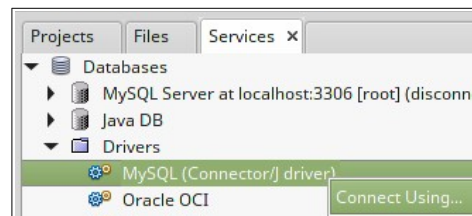


**NOTA:** Si estás utilizando la última versión de NetBeans es posible que en Services → Databases → Drivers ya te aparezca la opción 'MySQL (Connector/J driver)'. A pesar de ello puede que necesites instalar el driver. Dale al driver con clic derecho → 'Connect using' y comprueba si en la ventana te aparece el mensaje 'driver file is missing'. En tal caso, dale a 'add driver' y selecciona el archivo jar mencionado anteriormente.

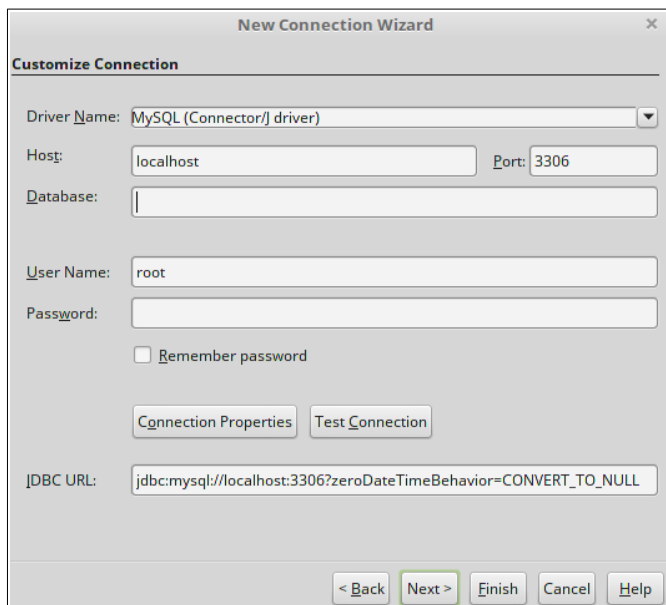
Llegados a este punto ya tenemos enlazado el driver JDBC de MySQL/MariaDB desde el Database Explorer. Ahora lo que nos queda es crear una conexión con una base de datos MySQL/MariaDB.

## 4.2 Conexión a Base de Datos mediante el JDBC Driver

Haz clic derecho en 'MySQL (Connector/J driver)' y selecciona la opción 'Connect Using...'.



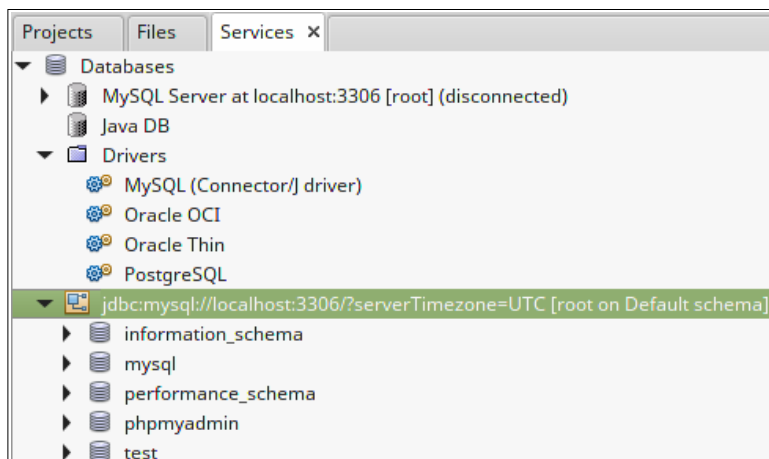
Aparecerá la ventana 'New Connection Wizard':



- **Driver Name:** El driver que queremos utilizar. En este caso 'MySQL (Connector/J driver)'.
- **Host:** La dirección IP o nombre de dominio de la máquina donde se encuentra instalado el servidor MySQL. En nuestro caso 'localhost' (máquina local).
- **Port:** Puerto de conexión. MySQL utiliza el 3306 por defecto.
- **Database:** El nombre de la base de datos a la que queremos acceder. Podemos dejar este campo vacío para poder acceder a todas las que existan en el servidor.
- **User Name:** Nombre de usuario. Utilizaremos 'root' (administrador).
- **Password:** Contraseña del usuario. Lo dejaremos en blanco ya que tras instalar MySQL la contraseña por defecto está vacía. En nuestro caso para practicar y aprender esto será suficiente, pero en un entorno real tendríamos que configurar una contraseña robusta.
- **JDBC URL:** La URL de conexión con el formato `jdbc:mysql://HOST:PUERTO`. Se crea automáticamente a partir de la información anterior. Es posible que incluya por defecto variables como 'zeroDateTimeBehaviour' en la imagen.

Haz clic en **Finalizar**. Si todo va bien, en Services → Databases aparecerá una nueva conexión llamada `jdbc:mysql://localhost:3306... [root on Default schema]`

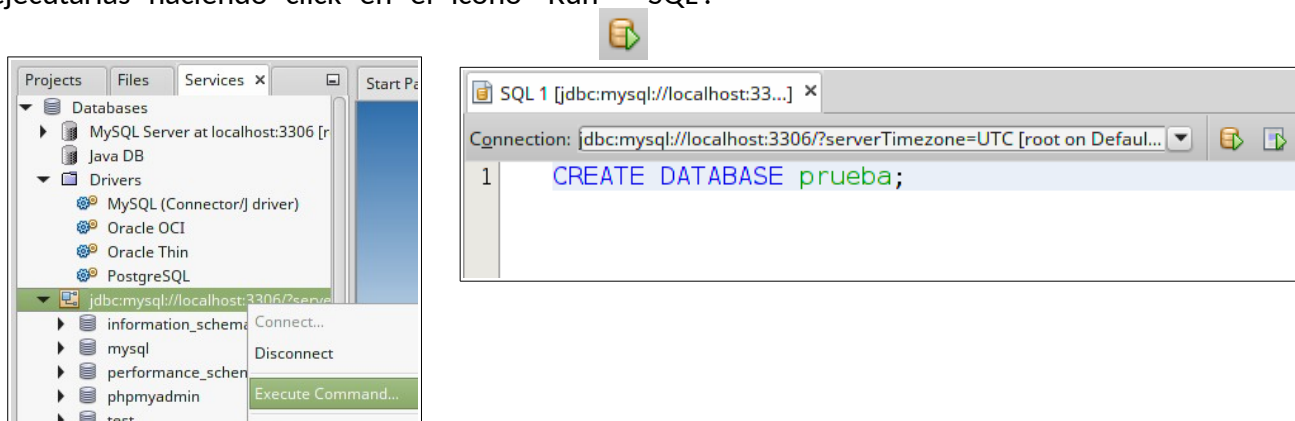
**NOTA:** Si falla debido a un error 'server time zone' puedes solucionarlo añadiendo al URL la variable `serverTimezone` con el valor UTC: `jdbc:mysql://localhost:3306/?serverTimezone=UTC`



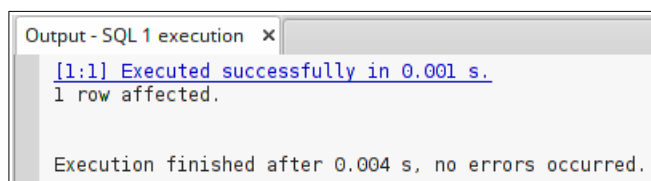
Si desplegamos la conexión creada veremos que podemos acceder a todas las bases de datos, tablas, etc. almacenadas en el servidor al que estamos conectados (a cada uno le aparecerán las bases de datos que tenga en su sistema, pueden ser diferentes a las de la imagen).

**Esto es muy útil ya que nos permitirá acceder cómodamente a la base de datos, realizar consultas, ejecutar comandos, etc. mientras desarrollamos un proyecto con NetBeans.**

Para ejecutar un comando sobre la base de datos a la que nos hemos conectado solo hay que hacer clic derecho sobre la conexión y seleccionar la opción 'Execute Command...'. Se abrirá un panel en el editor central en el que podremos introducir sentencias SQL (tantas como queramos) y ejecutarlas haciendo click en el icono 'Run SQL':



Prueba a ejecutar la sentencia **CREATE DATABASE prueba;** como en la imagen anterior. En el panel de abajo se mostrará un mensaje con el resultado de la ejecución.



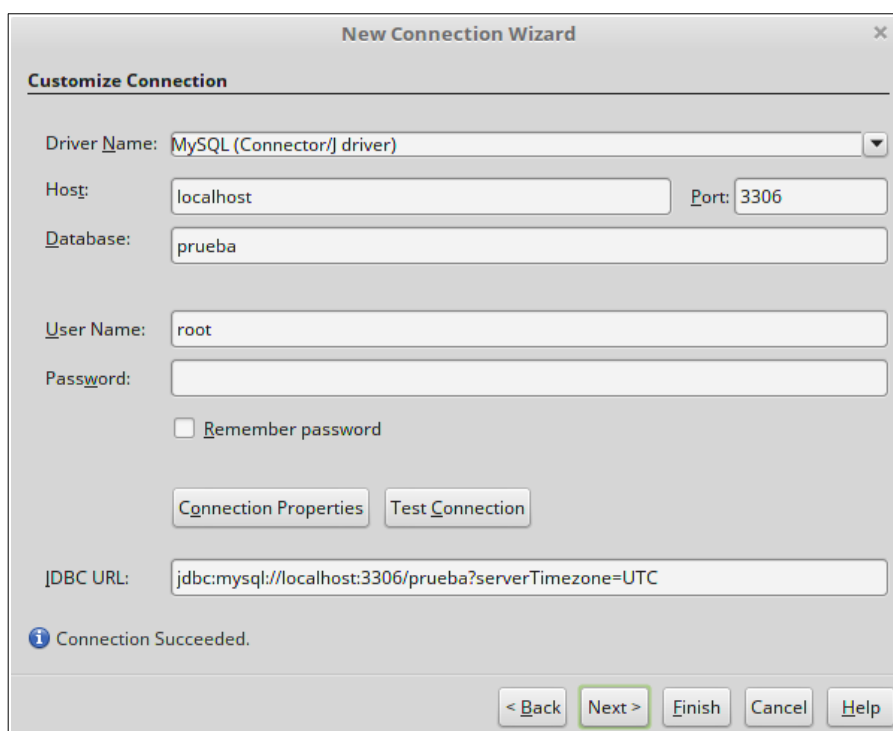
En este caso indica 'Executed successfully' (se ha ejecutado satisfactoriamente). En la conexión jdbc del panel de la izquierda aparecerá la nueva base de datos 'prueba' que hemos creado.

Hay que tener en cuenta que como hemos configurado esta conexión sin especificar ninguna base de datos concreta, si quisiéramos ejecutar sentencias sobre una base de datos de las disponibles tendríamos que utilizar siempre la sentencia `USE nombre_de_base_de_datos`. Por ejemplo:

```
USE tiendaonline;  
SELECT * FROM clientes;
```

Si vamos a trabajar habitualmente sobre la misma base de datos y queremos evitar tener que escribir siempre la sentencia `USE` en todas las ejecuciones de sentencias SQL, podemos crear una conexión configurada para utilizar una base de datos concreta. Para ello primero tendríamos que seguir el mismo procedimiento explicado en este apartado (Clic derecho en 'MySQL (Connector/J driver)' → 'Connect Using...' → Introducir los datos en el diálogo 'New Connection Wizard' y en el campo 'Database' escribir la base de datos deseada.

Por ejemplo, para el caso de la base de datos prueba que hemos creado, la conexión sería esta:



The screenshot shows the 'New Connection Wizard' dialog box with the 'Customize Connection' tab selected. The fields are filled as follows: Driver Name is 'MySQL (Connector/J driver)'; Host is 'localhost'; Port is '3306'; Database is 'prueba'; User Name is 'root'; Password is empty; 'Remember password' is unchecked; 'Connection Properties' and 'Test Connection' buttons are present; the JDBC URL is 'jdbc:mysql://localhost:3306/prueba?serverTimezone=UTC'; and a status message at the bottom says 'Connection Succeeded.' The 'Next >' button is highlighted with a green border.

La URL resultante sería: **`jdbc:mysql://localhost:3306/prueba?serverTimezone=UTC`**

En los siguientes apartados veremos algunos ejemplos de ejecución de sentencias SQL utilizando esta nueva conexión a la base de datos 'prueba'.

### 4.3 Crear una tabla

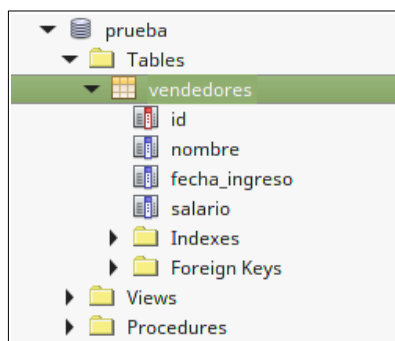
Desde el IDE de NetBeans podemos crear tablas de dos formas: por sentencias SQL o utilizando un asistente de creación de tablas.

#### 4.3.1 Mediante sentencias SQL

Seleccionamos la conexión `jdbc:mysql://localhost:3306/prueba` creada en el apartado anterior. Hacemos clic derecho sobre ella → *Execute Command...* e introducimos el siguiente comando SQL:

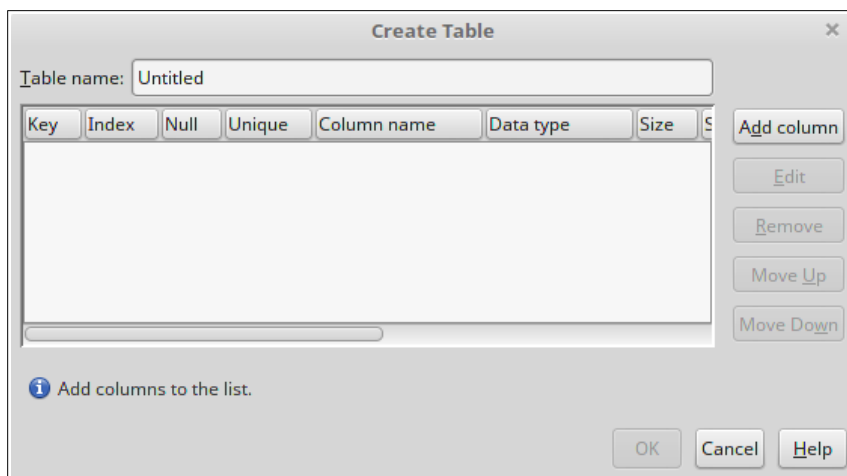
```
CREATE TABLE vendedores (  
  id int NOT NULL auto_increment,  
  nombre varchar(50) NOT NULL default '',  
  fecha_ingreso date NOT NULL default '0000-00-00',  
  salario float NOT NULL default '0',  
  PRIMARY KEY (id) );
```

Si todo ha ido bien dentro de 'prueba → Tables' veremos la nueva tabla. Es posible que necesitemos hacer clic derecho → *Refresh* (sobre prueba) para que se actualice la información.



#### 4.3.2 Mediante el asistente de creación de tablas

Sobre la carpeta 'Tables' de la conexión deseada, hacemos clic derecho y elegimos 'Create Table'. Se abrirá el asistente de creación de tablas. Añadimos todas las columnas que necesitemos mediante la opción 'Add Column' (indicando las opciones de cada columna) y hacemos clic en 'OK'.

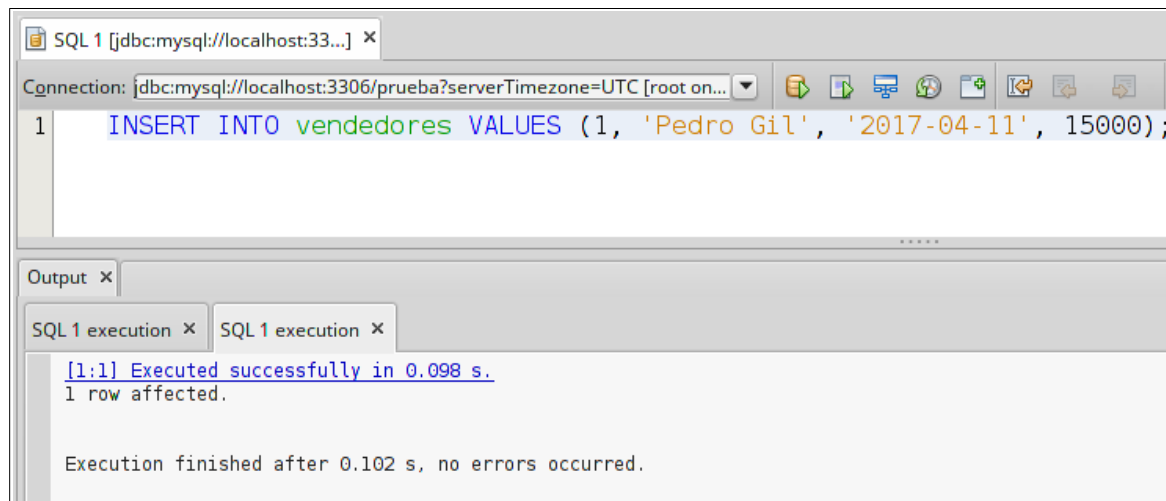




#### 4.4 Insertar datos

Ejecutar el comando INSERT INTO... sobre la conexión deseada. Por ejemplo:

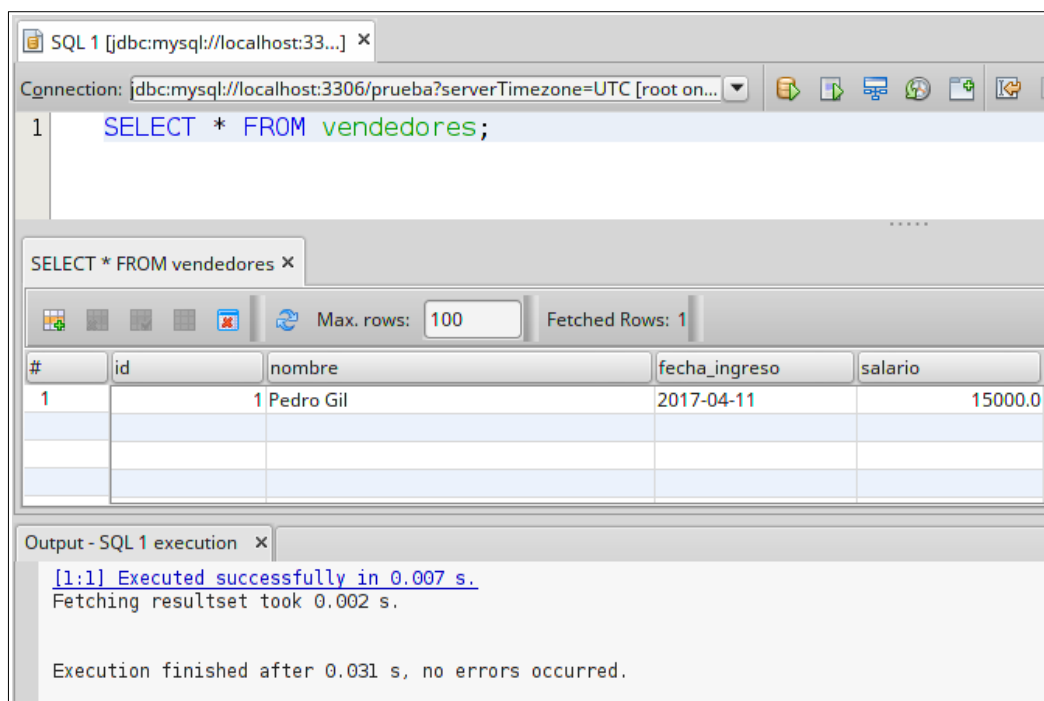
```
INSERT INTO vendedores VALUES (1, 'Pedro Gil', '2017-04-11', 15000);
```



#### 4.5 Consultar valores

Ejecutar el comando SELECT... sobre la conexión deseada. Por ejemplo:

```
SELECT * FROM vendedores;
```

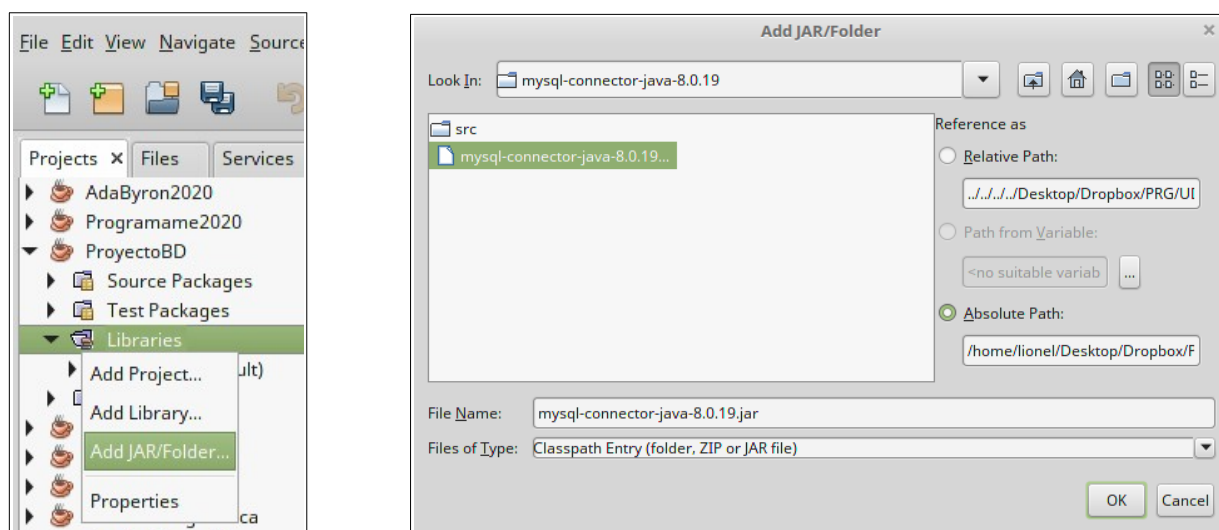


## 5. ACCESO A BASES DE DATOS MEDIANTE CÓDIGO JAVA

En este apartado se ofrece una introducción a los aspectos fundamentales del acceso a bases de datos mediante código Java. En los siguientes apartados se explicarán algunos aspectos en mayor detalle, sobre todo los relacionados con las clases Statement y ResultSet.

### 5.1 Añadir la librería JDBC al proyecto

Para poder utilizar la librería JDBC en un proyecto Java primero deberemos añadirla al proyecto. Para ello debemos hacer clic derecho sobre la carpeta 'Libraries' del proyecto y seleccionar 'Add JAR/Folder'. En la ventana emergente deberemos seleccionar el archivo del driver previamente descargado mysql-connector-java-8.0.19.jar y clic en OK.



### 5.2 Cargar el Driver

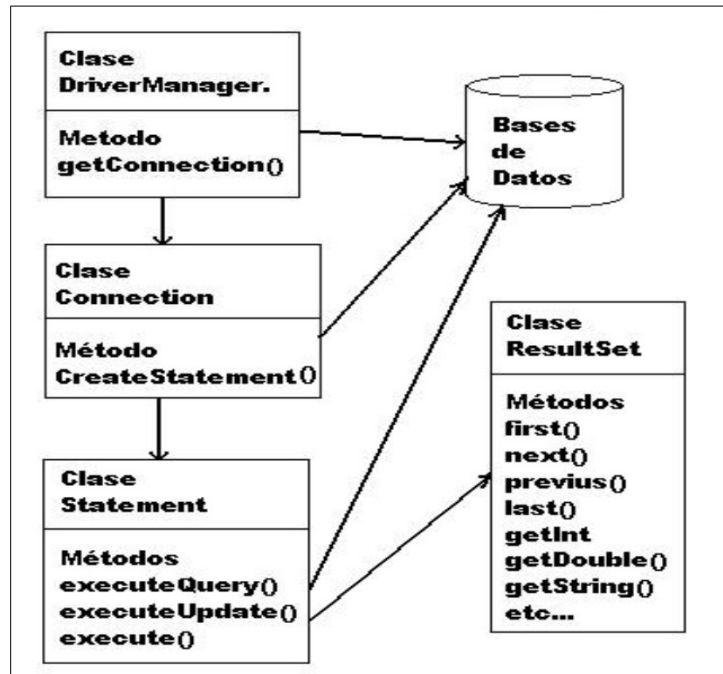
En un proyecto Java que realice conexiones a bases de datos es necesario, antes que nada, utilizar `Class.forName(...).newInstance()` para cargar dinámicamente el Driver que vamos a utilizar. Esto solo es necesario hacerlo una vez en nuestro programa. Puede lanzar excepciones por lo que es necesario utilizar un bloque try-catch.

```
try {  
    Class.forName("com.mysql.cj.jdbc.Driver").newInstance();  
} catch (Exception e) {  
    // manejamos el error  
}
```

Hay que tener en cuenta que las clases y métodos utilizados para conectarse a una base de datos (explicados más adelante) funcionan con todos los drivers disponibles para Java (JDBC es solo uno, hay muchos más). Esto es posible ya que el estándar de Java solo los define como interfaces (interface) y cada librería driver los implementa (define las clases y su código). Por ello es necesario utilizar `Class.forName(...)` para indicarle a Java qué driver vamos a utilizar.

Este nivel de abstracción facilita el desarrollo de proyectos ya que si necesitáramos utilizar otro sistema de base de datos (que no fuera MySQL) solo necesitaríamos cambiar la línea de código que carga el driver y poco más. Si cada sistema de base de datos necesitara que utilizáramos distintas clases y métodos todo sería mucho más complicado.

Las cuatro clases fundamentales que toda aplicación Java necesita para conectarse a una base de datos y ejecutar sentencias son: **DriverManager**, **Connection**, **Statement** y **ResultSet**.



### 5.3 Clase DriverManager

La clase **java.sql.DriverManager** es la capa gestora del driver JDBC. Se encarga de manejar el Driver apropiado y **permite crear conexiones con una base de datos** mediante el método estático **getConnection(...)** que tiene dos variantes:

```

DriverManager.getConnection(String url)
DriverManager.getConnection(String url, String user, String password)

```

Este método intentará establecer una conexión con la base de datos según la URL indicada. Opcionalmente se le puede pasar el usuario y contraseña como argumento (también se puede indicar en la propia URL). Si la conexión es satisfactoria devolverá un objeto **Connection**.

Ejemplo de conexión a la base de datos 'prueba' en localhost:

```

String url = "jdbc:mysql://localhost:3306/prueba";
Connection conn = DriverManager.getConnection(url, "root", "");

```

Este método puede lanzar dos tipos de excepciones (que habrá que manejar con un try-catch):

- **SQLException**: La conexión no ha podido producirse. Puede ser por multitud de motivos como una URL mal formada, un error en la red, host o puerto incorrecto, base de datos no existente, usuario y contraseña no válidos, etc.
- **SQLException<sup>2</sup>**: Se ha superado el LoginTimeout sin recibir respuesta del servidor.

<sup>2</sup> Es una subclase de SQLException, así que para capturarla, hay que hacerlo previamente al catch de SQLException.

## 5.4 Clase Connection

Un objeto **java.sql.Connection** representa una sesión de conexión con una base de datos. Una aplicación puede tener tantas conexiones como necesite, ya sea con una o varias bases de datos.

El método más relevante es **createStatement()** que devuelve un objeto **Statement** asociado a dicha conexión que permite ejecutar sentencias SQL. El método **createStatement()** puede lanzar excepciones de tipo **SQLException**.

```
Statement stmt = conn.createStatement();
```

Cuando ya no la necesitemos es aconsejable **cerrar la conexión con close()** para liberar recursos.

```
conn.close();
```

## 5.5 Clase Statement

Un objeto **java.sql.Statement** permite **ejecutar sentencias SQL en la base de datos** a través de la conexión con la que se creó el **Statement** (ver apartado anterior). Los tres métodos más comunes de ejecución de sentencias SQL son **executeQuery(...)**, **executeUpdate(...)** y **execute(...)**. Pueden lanzar excepciones de tipo **SQLException** y **SQLTimeoutException**.

- **ResultSet executeQuery(String sql):** Ejecuta la sentencia sql indicada (de tipo SELECT). Devuelve un objeto **ResultSet** con los datos proporcionados por el servidor.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM vendedores");
```

- **int executeUpdate(String sql):** Ejecuta la sentencia sql indicada (de tipo DML como por ejemplo INSERT, UPDATE o DELETE). Devuelve un el número de registros que han sido insertados, modificados o eliminados.

```
int nr = stmt.executeUpdate("INSERT INTO vendedores VALUES (1,  
                        'Pedro Gil', '2017-04-11', 15000);")
```

Cuando ya no lo necesitemos es aconsejable **cerrar el statement con close()** para liberar recursos.

```
stmt.close();
```

## 5.6 Clase ResultSet

Un objeto **java.sql.ResultSet** contiene un conjunto de resultados (datos) obtenidos tras ejecutar una sentencia SQL, normalmente de tipo SELECT. Es una **estructura de datos en forma de tabla** con **registros (filas)** que podemos recorrer para acceder a la información de sus **campos (columnas)**.

ResultSet utiliza internamente un cursor que apunta al 'registro actual' sobre el que podemos operar. Inicialmente dicho cursor está situado antes de la primera fila y disponemos de varios métodos para desplazar el cursor. El más común es next():

- **boolean next():** Mueve el cursor al siguiente registro. Devuelve true si fué posible y false en caso contrario (si ya llegamos al final de la tabla).

Algunos de los métodos para obtener los datos del registro actual son:

- **String getString(String columnLabel):** Devuelve un dato String de la columna indicada por su nombre. Por ejemplo: `rs.getString("nombre")`
- **String getString(int columnIndex):** Devuelve un dato String de la columna indicada por su nombre. La primera columna es la 1, no la cero. Por ejemplo: `rs.getString(2)`

Existen métodos análogos a los anteriores para obtener valores de tipo int, long, float, double, boolean, Date, Time, Array, etc. Pueden consultarse todos en la [documentación oficial de Java](#).

- |   |  |
|---|--|
| • <b>int getInt(String columnLabel)</b>         | <b>int getInt(int columnIndex)</b>         |
| • <b>double getDouble(String columnLabel)</b>   | <b>double getDouble(int columnIndex)</b>   |
| • <b>boolean getBoolean(String columnLabel)</b> | <b>boolean getBoolean(int columnIndex)</b> |
| • <b>Date getDate(String columnLabel)</b>       | <b>int getDate(int columnIndex)</b>        |
| • etc.  |  |

Más adelante veremos cómo se realiza la modificación e inserción de datos.

Todos estos métodos pueden lanzar una **SQLException**.

Veamos un ejemplo de cómo recorrer un ResultSet llamado rs y mostrarlo por pantalla:

```
while(rs.next()) {  
    int id      = rs.getInt("id");  
    String nombre = rs.getString("nombre");  
    Date fecha   = rs.getDate("fecha_ingreso");  
    float salario = rs.getFloat("salario");  
    System.out.println(id + " " + nombre + " " + fecha + " " + salario);  
}
```

## 5.7 Ejemplo completo

Veamos un ejemplo completo de conexión y acceso a una base de datos utilizando todos los elementos mencionados en este apartado.

```
try {
    // Cargamos la clase que implementa el Driver
    Class.forName("com.mysql.cj.jdbc.Driver").newInstance();

    // Creamos una nueva conexión a la base de datos 'prueba'
    String url = "jdbc:mysql://localhost:3306/prueba?serverTimezone=UTC";
    Connection conn = DriverManager.getConnection(url,"root","");

    // Obtenemos un Statement de la conexión
    Statement st = conn.createStatement();

    // Ejecutamos una consulta SELECT para obtener la tabla vendedores
    String sql = "SELECT * FROM vendedores";
    ResultSet rs = st.executeQuery(sql);

    // Recorremos todo el ResultSet y mostramos sus datos
    while(rs.next()) {
        int id          = rs.getInt("id");
        String nombre   = rs.getString("nombre");
        Date fecha      = rs.getDate("fecha_ingreso");
        float salario   = rs.getFloat("salario");
        System.out.println(id + " " + nombre + " " + fecha + " " + salario);
    }

    // Cerramos el statement y la conexión
    st.close();
    conn.close();

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

## 6. NAVEGABILIDAD Y CONCURRENCIA

Cuando invocamos a **`createStatement()`** sin argumentos, como hemos visto anteriormente, al ejecutar sentencias SQL obtendremos un **ResultSet por defecto en el que el cursor solo puede moverse hacia adelante y los datos son de solo lectura**. A veces esto no es suficiente y necesitamos mayor funcionalidad.

Por ello el método **`createStatement()`** está sobrecargado (existen varias versiones de dicho método) lo cual nos permite invocarlo con argumentos en los que podemos especificar el funcionamiento.

- **Statement `createStatement(int resultSetType, int resultSetConcurrency)`**: Devuelve un objeto Statement cuyos objetos ResultSet serán del tipo y concurrencia especificados. Los valores válidos son constantes definidas en ResultSet.

El argumento **`resultSetType`** indica el tipo de ResultSet:

- **ResultSet.TYPE\_FORWARD\_ONLY**: ResultSet por defecto, forward-only y no-actualizable.
  - Solo permite movimiento hacia delante con `next()`.
  - Sus datos NO se actualizan. Es decir, no reflejará cambios producidos en la base de datos. Contiene una instantánea del momento en el que se realizó la consulta.
- **ResultSet.TYPE\_SCROLL\_INSENSITIVE**: ResultSet desplazable y no actualizable.
  - Permite libertad de movimiento del cursor con otros métodos como `first()`, `previous()`, `last()`, etc. además de `next()`.
  - Sus datos NO se actualizan, como en el caso anterior.
- **ResultSet.TYPE\_SCROLL\_SENSITIVE**: ResultSet desplazable y actualizable.
  - Permite libertad de movimientos del cursor, como en el caso anterior.
  - Sus datos SÍ se actualizan. Es decir, mientras el ResultSet esté abierto se actualizará automáticamente con los cambios producidos en la base de datos. Esto puede suceder incluso mientras se está recorriendo el ResultSet, lo cual puede ser conveniente o contraproducente según el caso.

El argumento **`resultSetConcurrency`** indica la concurrencia del ResultSet:

- **ResultSet.CONCUR\_READ\_ONLY**: Solo lectura. Es el valor por defecto.
- **ResultSet.CONCUR\_UPDATABLE**: Permite modificar los datos almacenados en el ResultSet para luego aplicar los cambios sobre la base de datos (más adelante se verá cómo).

El ResultSet por defecto que se obtiene con `createStatement()` sin argumentos es el mismo que con `createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY)`.

## 7. CONSULTAS

### 7.1 Navegación de un ResultSet

Como ya se ha visto, 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 métodos de navegación por los registros como `next()` que desplaza el curso al siguiente registro del *ResultSet*.

Además de este método de desplazamiento básico, existen otros de desplazamiento libre que podremos utilizar siempre y cuando el *ResultSet* sea de tipo *ResultSet.TYPE\_SCROLL\_INSENSITIVE* o *ResultSet.TYPE\_SCROLL\_SENSITIVE* como se ha dicho antes.

Algunos de estos métodos son:

- **void beforeFirst():** Mueve el cursor antes de la primera fila.
- **boolean first():** Mueve el cursor a la primera fila.
- **boolean next():** Mueve el cursor a la siguiente fila. Permitido en todos los tipos de *ResultSet*.
- **boolean previous():** Mueve el cursor a la fila anterior.
- **boolean last():** Mueve el cursor a la última fila.
- **void afterLast():** Mover el cursor después de la última fila.
- **boolean absolute(int row):** Posiciona el cursor en el número de registro indicado. Hay que tener en cuenta que el primer registro es el 1, no el cero. Por ejemplo `absolute(7)` desplazará el cursor al séptimo registro. Si valor es negativo se posiciona en el número de registro indicado pero empezando a contar desde el final (el último es el -1). Por ejemplo si tiene 10 registros y llamamos `absolute(-2)` se desplazará al registro n.º 9.
- **boolean relative(int registros):** Desplaza el cursor un número relativo de registros, que puede ser positivo o negativo. Por ejemplo si el cursor está en el registro 5 y llamamos a `relative(10)` se desplazará al registro 15. Si luego llamamos a `relative(-4)` se desplazará al registro 11.

Los métodos que devuelven un tipo boolean devolverán 'true' si ha sido posible mover el cursor a un registro válido, y 'false' en caso contrario, por ejemplo si no tiene ningún registro o hemos saltado a un número de registro que no existe.

Todos estos métodos pueden producir una excepción de tipo *SQLException*.

También existen otros métodos relacionados con la posición del cursor.

- **int getRow():** Devuelve el número de registro actual. Cero si no hay registro actual.
- **boolean isBeforeFirst():** Devuelve 'true' si el cursor está antes del primer registro.
- **boolean isFirst():** Devuelve 'true' si el cursor está en el primer registro.
- **boolean isLast():** Devuelve 'true' si el cursor está en el último registro.
- **boolean isAfterLast():** Devuelve 'true' si el cursor está después del último registro.



## 7.2 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*. No es necesario que las columnas sean obtenidas utilizando un orden determinado.

Para designar una columna podemos utilizar su nombre o bien su número (empezando por 1).

Por ejemplo si la segunda columna de un objeto *ResultSet* se llama "título" y almacena datos de tipo *String*, se podrá recuperar su valor de las dos formas siguientes:

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

Es importante tener en cuenta que las columnas se numeran de izquierda a derecha y que la primera es la número 1, no la cero. También que las columnas no son case sensitive, es decir, no distinguen entre mayúsculas y minúsculas.



La información referente a las columnas de un *ResultSet* se puede obtener llamando al **método `getMetaData()`** que 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.

## 7.3 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* de tipo SQL a un objeto *String* de Java.

Algo parecido sucede con otros tipos de datos SQL como por ejemplo *DATE*. Podremos acceder a él tanto con `getDate()` como con `getString()`. La diferencia es que el primero devolverá un objeto Java de tipo *Date* y el segundo devolverá un *String*.

Siempre que sea posible el driver JDBC convertirá el tipo de dato almacenado en la base de datos al tipo solicitado por el método `getXXX()`, pero hay conversiones que no se pueden realizar y lanzarán una excepción, como por ejemplo si intentamos hacer un `getInt()` sobre un campo que no contiene un valor numérico.

## 8. MODIFICACIÓN

Para poder modificar los datos que contiene un *ResultSet* necesitamos un *ResultSet* de tipo modificable. Para ello debemos utilizar la constante *ResultSet.CONCUR\_UPDATABLE* al llamar al método *createStatement()* como se ha visto antes.

**Para modificar los valores de un registro existente se utilizan una serie de métodos *updateXXX()* de *ResultSet*.** Las XXX indican el tipo del dato y hay tantos distintos como sucede con los métodos *getXXX()* de este mismo interfaz: ***updateString()*, *updateInt()*, *updateDouble()*, *updateDate()*, etc.**

La diferencia es que **los métodos *updateXXX()* necesitan dos argumentos:**

- **La columna que deseamos actualizar** (por su nombre o por su número de columna).
- **El valor que queremos almacenar** en dicha columna (del tipo que sea).

Por ejemplo para modificar el campo 'edad' almacenando el entero 28 habría que llamar al siguiente método, suponiendo que *rs* es un objeto *ResultSet*:

```
rs.updateInt("edad", 28);
```

También podría hacerse de la siguiente manera, suponiendo que la columna "edad" es la segunda:

```
rs.updateInt(2, 28);
```

Los métodos *updateXXX()* no devuelven ningún valor (son de tipo void). Si se produce algún error se lanzará una *SQLException*.

Posteriormente hay que **llamar a *updateRow()* para que los cambios realizados se apliquen sobre la base de datos**. El Driver JDBC se encargará de ejecutar las sentencias SQL necesarias. Esta es una característica muy potente ya que nos facilita enormemente la tarea de modificar los datos de una base de datos. Este método devuelve void.

En resumen, el proceso para realizar la modificación de una fila de un *ResultSet* es el siguiente:

1. **Desplazamos el cursor al registro** que queremos modificar.
2. Llamamos a todos los métodos ***updateXXX(...)*** que necesitamos.
3. Llamamos a ***updateRow()*** para que los cambios se apliquen a la base de datos.

Es importante entender que **hay que llamar a *updateRow()* antes de desplazar el cursor**. Si desplazamos el cursor antes de llamar a *updateRow()*, se perderán los cambios.

Si queremos **cancelar las modificaciones de un registro del *ResultSet*** podemos llamar a ***cancelRowUpdates()***, que cancela todas las modificaciones realizadas sobre el registro actual.

Si ya hemos llamado a *updateRow()* el método *cancelRowUpdates()* no tendrá ningún efecto.

El siguiente código de ejemplo muestra cómo modificar el campo 'dirección' del último registro de un *ResultSet* que contiene el resultado de una SELECT sobre la tabla de clientes. Supondremos que *conn* es un objeto *Connection* previamente creado:

```
// Creamos un Statement scrollable y modificable
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                      ResultSet.CONCUR_UPDATABLE);

// Ejecutamos un SELECT y obtenemos la tabla clientes en un ResultSet
String sql = "SELECT * FROM clientes";
ResultSet rs = stmt.executeQuery(sql);

// Vamos al último registro, lo modificamos y actualizamos la base de datos
rs.last();
rs.updateString("direccion", "C/ Pepe Ciges, 3");
rs.updateRow();
```

## 9. INSERCIÓN

Para insertar nuevos registros necesitaremos utilizar, al menos, estos dos métodos:

- **void moveToInsertRow():** Desplaza el cursor al 'registro de inserción'. Es un registro especial utilizado para insertar nuevos registros en el ResultSet. Posteriormente tendremos que llamar a los métodos updateXXX() ya conocidos para establecer los valores del registro de inserción. Para finalizar hay que llamar a insertRow().
- **void insertRow():** Inserta el 'registro de inserción' en el ResultSet, pasando a ser un registro normal más, y también lo inserta en la base de datos.

El siguiente código inserta un nuevo registro en la tabla 'clientes'. Supondremos que `conn` es un objeto Connection previamente creado:

```
// Creamos un Statement scrollable y modificable
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                       ResultSet.CONCUR_UPDATABLE);

// Ejecutamos un SELECT y obtenemos la tabla clientes en un ResultSet
String sql = "SELECT * FROM clientes";
ResultSet rs = stmt.executeQuery(sql);

// Creamos un nuevo registro y lo insertamos
rs.moveToInsertRow();
rs.updateString(2, "Killy Lopez");
rs.updateString(3, "Wall Street 3674");
rs.insertRow();
```

Los campos cuyo valor no se haya establecido con updateXXX() tendrán un valor NULL. Si en la base de datos dicho campo no está configurado para admitir nulos se producirá una SQLException.

Tras insertar nuestro nuevo registro en el objeto ResultSet podremos volver a la anterior posición en la que se encontraba el cursor (antes de invocar moveToInsertRow()) llamando al método moveToCurrentRow(). Este método sólo se puede utilizar en combinación con moveToInsertRow().

## 10. BORRADO

Para eliminar un registro solo hay que desplazar el cursor al registro deseado y llamar al método:

- **void deleteRow():** Elimina el registro actual del ResultSet y también de la base de datos.

El siguiente código borra el tercer registro de la tabla 'clientes':

```
// Creamos un Statement scrollable y modificable
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                       ResultSet.CONCUR_UPDATABLE);

// Ejecutamos un SELECT y obtenemos la tabla clientes en un ResultSet
String sql = "SELECT * FROM clientes";
ResultSet rs = stmt.executeQuery(sql);

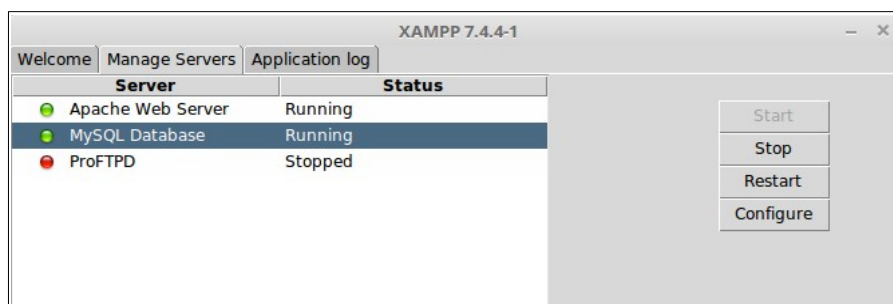
// Desplazamos el cursor al tercer registro
rs.absolute(3)
rs.deleteRow();
```

## 11. PROGRAMA GESTOR DE CLIENTES

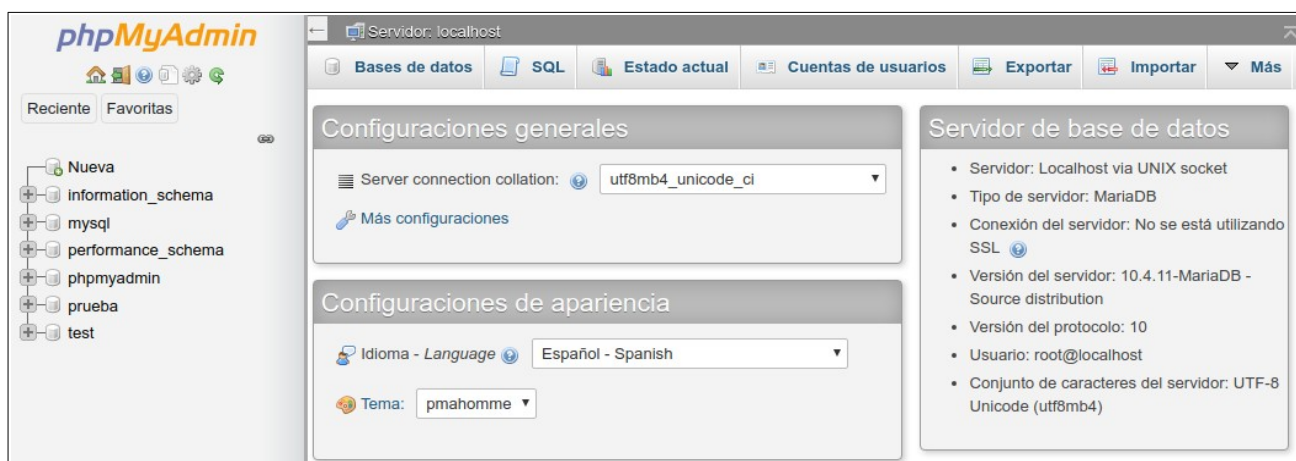
Veamos un ejemplo de un programa de gestión de clientes de una tienda, con interfaz gráfica de usuario y acceso a base de datos MySQL. Este programa permitirá consultar y modificar la información de los clientes así como darlos de alta de y de baja. El código del proyecto y la base de datos pueden contrarse en el aula virtual.

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 al crear nuevos registros no es necesario darle un valor.

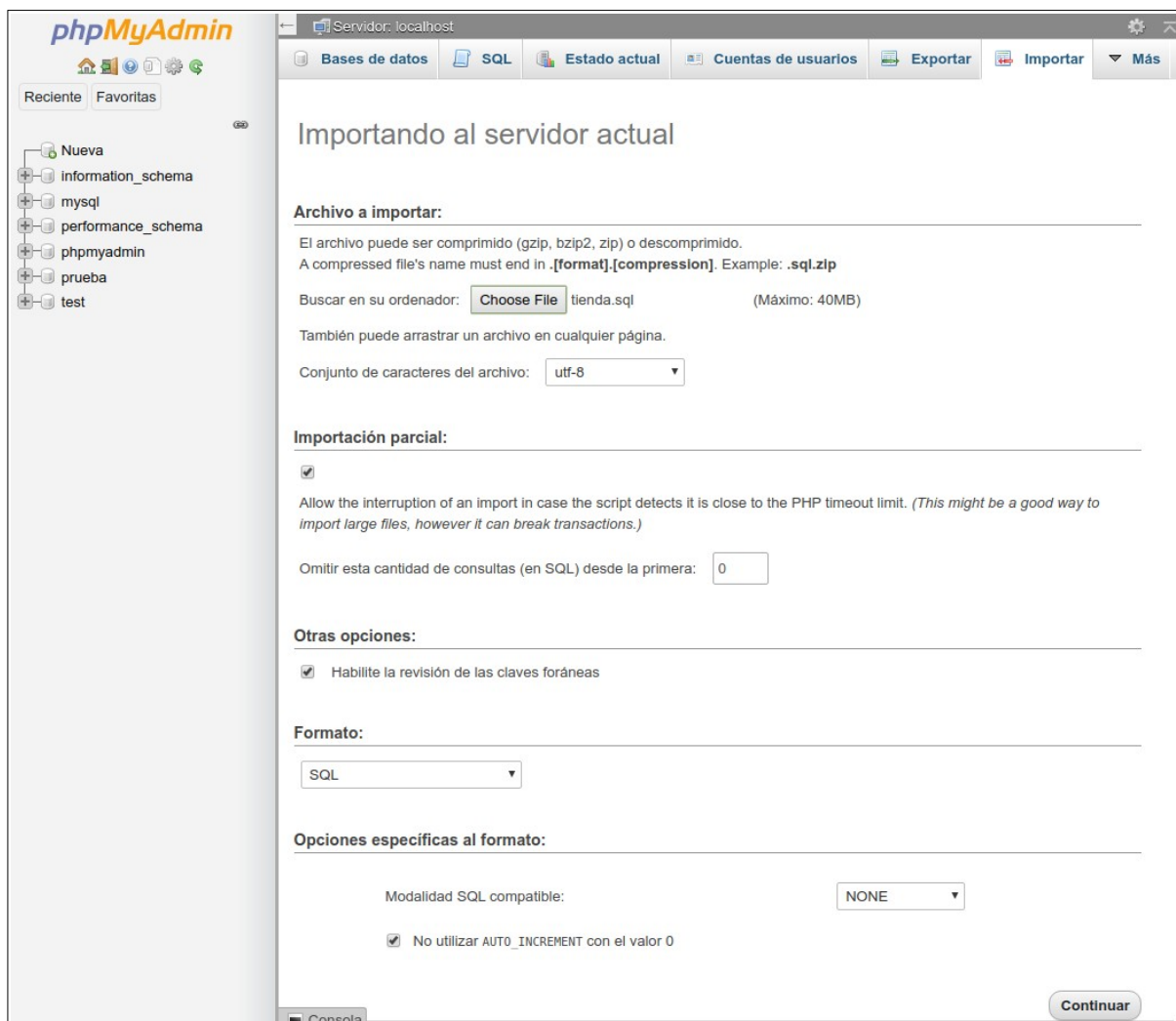
Primero necesitamos importar la base de datos a nuestro servidor MySQL. Abrimos el panel de control XAMPP y nos aseguramos de que *Apache* y *MySQL* están funcionando. En caso contrario los iniciamos.



Abrimos un navegador web, vamos a <http://localhost> y accedemos a phpMyAdmin. También podemos acceder directamente con <http://localhost/phpmyadmin/>. Esta aplicación web alojada en nuestro servidor Apache nos permite interactuar con el servidor MySQL de una forma sencilla.

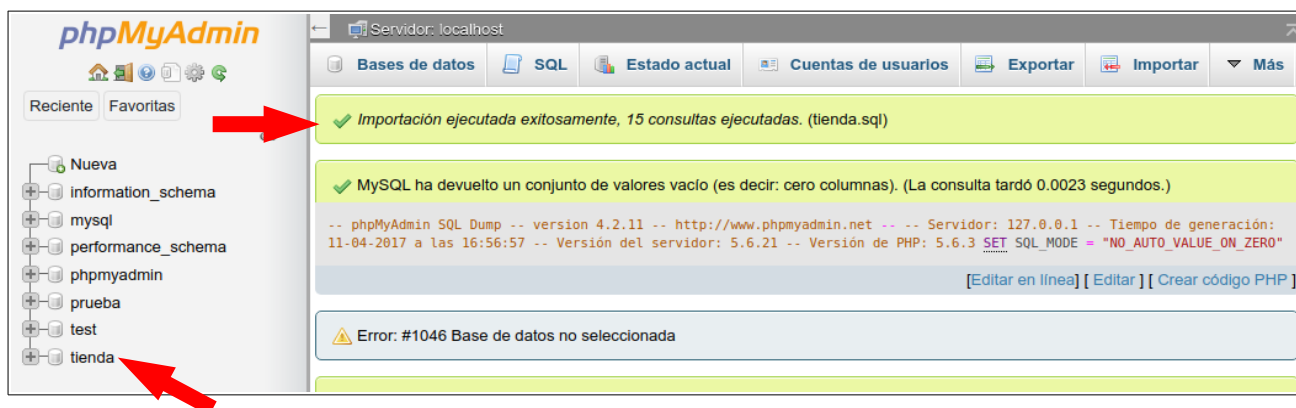


A continuación hacemos clic en 'Importar' y seleccionamos el archivo 'tienda.sql' que podéis descargar del aula virtual. Este archivo contiene las sentencias sql para crear la base de datos que necesitamos.



The screenshot shows the phpMyAdmin interface for importing a file. The left sidebar lists databases: Nueva, information\_schema, mysql, performance\_schema, phpmyadmin, prueba, and test. The main panel is titled 'Importando al servidor actual'. It includes sections for 'Archivo a importar' (with a 'Choose File' button and a file name 'tienda.sql'), 'Importación parcial' (with a checkbox for allowing interruption), 'Otras opciones' (with a checkbox for foreign key checks), 'Formato' (set to 'SQL'), and 'Opciones específicas al formato' (with a checkbox for 'No utilizar AUTO\_INCREMENT'). A 'Continuar' button is at the bottom right.

Hacemos clic en 'Continuar' y si todo va bien la base de datos 'tienda' se importará correctamente.



This screenshot shows the phpMyAdmin interface after the import. The left sidebar has a red arrow pointing to the 'tienda' database. The main panel displays a green success message: 'Importación ejecutada exitosamente, 15 consultas ejecutadas. (tienda.sql)'. Below it, another green message states: 'MySQL ha devuelto un conjunto de valores vacío (es decir: cero columnas). (La consulta tardó 0.0023 segundos.)'. A SQL dump is visible, including version information and the 'SET SQL\_MODE = "NO\_AUTO\_VALUE\_ON\_ZERO"' statement. At the bottom, a blue error message reads: 'Error: #1046 Base de datos no seleccionada'.

Ya tenemos la base de datos en nuestro servidor MySQL.

El siguiente paso es desarrollar la aplicación. Deberemos crear un nuevo proyecto Java e importar el JDBC connector como una librería JAR tal y como se ha explicado al principio de esta unidad.

A continuación creamos un JFrame llamado **VentanaClientes.java** y diseñamos la interfaz gráfica. Por ejemplo así:



El código relacionado con el acceso a la base de datos estará en una clase distinta llamada **BDClientes.java**. Esta clase contendrá varios métodos públicos para las distintas operaciones que necesitaremos utilizar desde la interfaz gráfica: conectar a la base de datos, consultar el listado de clientes, así como insertar, actualizar y borrar clientes.

Es decir, la clase VentanaClientes contendrá el código de la interfaz gráfica (JFrame) y utilizará la clase BDClientes para las operaciones de acceso a la base de datos. Esta división de 'responsabilidades' facilita el desarrollo y hace que nuestro código sea más modular.

En el aula virtual podéis encontrar el código a modo de ejemplo.



## 12. 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.