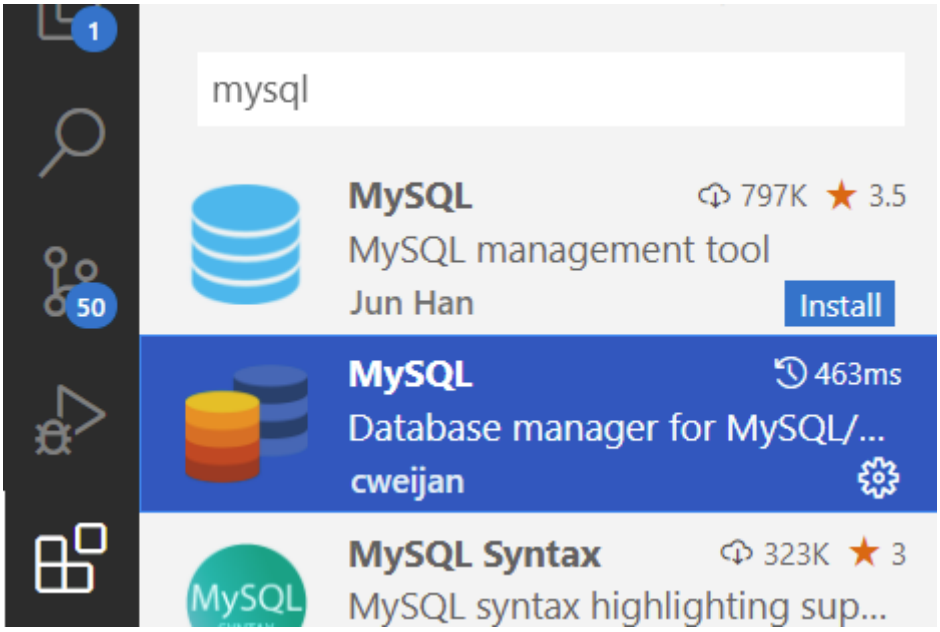
# Acceso a MySQL




En esta sesión trataremos cómo conectar con un sistema SQL tradicional, como MariaDB/MySQL. En primer lugar, daremos unas nociones básicas sobre cómo acceder a la base de datos a través de alguna herramienta que nos facilite su gestión, y después veremos qué librería(s) utilizar en Node.js para acceder a estos sistemas de bases de datos.

## 1. Gestión de bases de datos MariaDB/MySQL

Para trabajar con bases de datos MariaDB y poderlas gestionar cómodamente, vamos a utilizar una extensión del IDE Visual Studio Code que ya hemos instalado en sesiones previas, llamada *MySQL*.

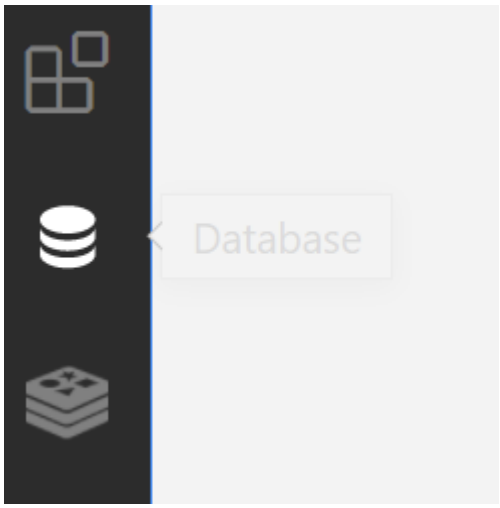



Nos va a permitir conectar tanto a sistemas MySQL como a otros sistemas de otras bases de datos, tanto relacionales como No-SQL.

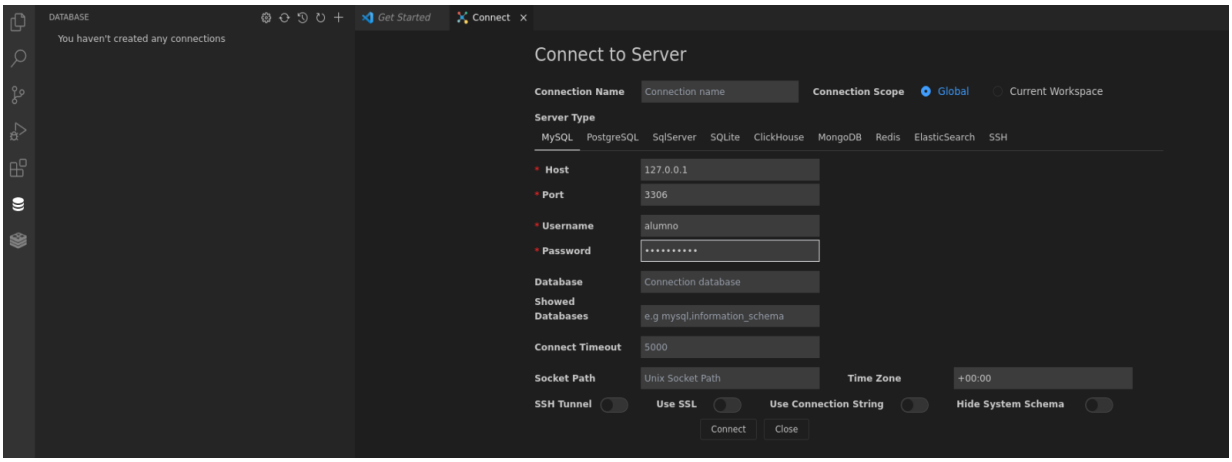
### 1.1. Configuración de la extensión *MySQL*

> **NOTA**: en la máquina virtual completa que se proporciona para este curso ya está la extensión instalada y configurada. Y si has seguido los pasos dados en el documento de instalación para hacerlo tú mismo, también la tendrás lista. Puedes ir directamente al apartado siguiente para ver cómo utilizarla.

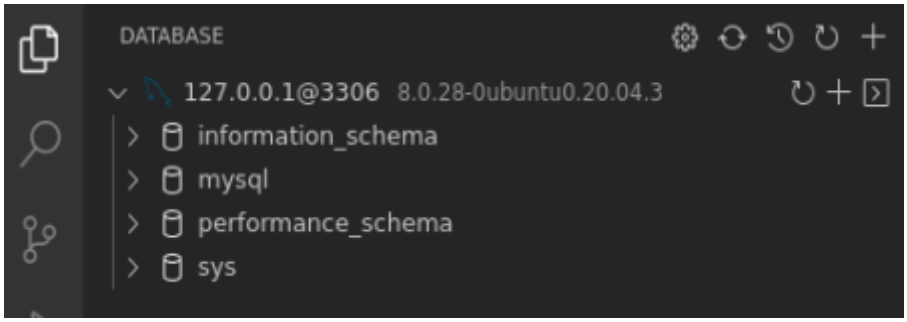
Antes de poder gestionar cualquier base de datos, debemos establecer una conexión con el servidor. Haremos clic en el icono de *Databases* de la barra izquierda:




Después, debemos crear la conexión (si no la tenemos hecha ya), haciendo clic en el botón "+" de la parte superior izquierda.




Tras conectar, podremos ver en el panel izquierdo las bases de datos existentes:



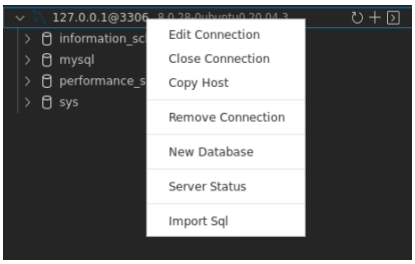

## 1.2. Uso básico de la extensión *MySQL*

Desde la extensión *MySQL* podemos hacer las tareas habituales de gestión de una base de datos, tales como:

- Crear bases de datos

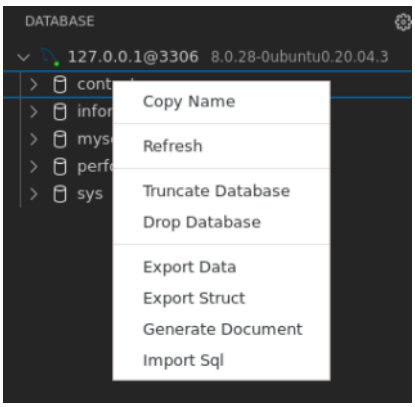
- Importar/Exportar bases de datos desde/a scripts de *backup*
- Crear tablas con sus campos
- ... etc.

Para crear una base de datos, hacemos clic derecho sobre la conexión (panel izquierdo) y elegimos la opción correspondiente:



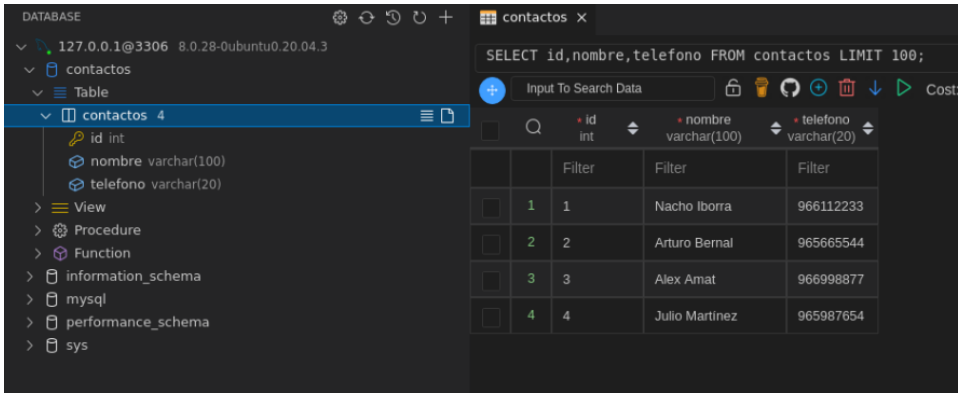

En la instrucción SQL que aparece en el panel derecho, completamos el nombre de la base de datos y pulsamos en el icono de *play* en la parte superior derecha para ejecutar la instrucción y crear la base de datos.


Para rellenar el contenido de la base de datos creada, podemos importar las instrucciones de algún *script* SQL. Para ello hacemos clic derecho sobre la base de datos y elegimos la opción de importar del menú contextual:

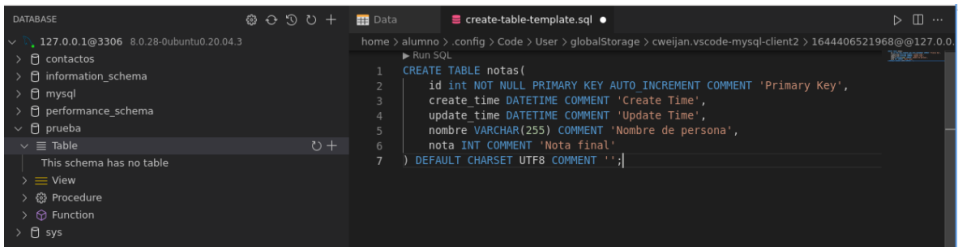



> **NOTA**: en ocasiones el propio *script* SQL ya tiene la instrucción *CREATE DATABASE*, con lo que no es necesario crearla antes de importar. Pero puede ser una buena práctica para evitar que dé error. A fin de cuentas, si la base de datos ya existe, esta instrucción no la creará de nuevo.

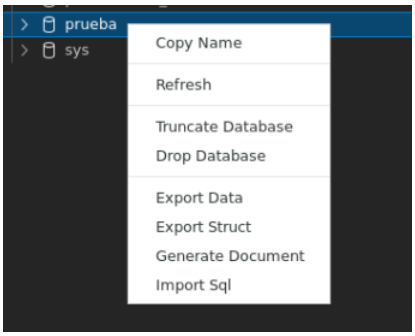
Una vez tengamos la base de datos importada, haciendo doble clic en cada tabla podemos ver su contenido en el panel derecho:




En ocasiones también nos puede interesar crear una base de datos vacía y definir sus tablas desde cero. En ese caso, desplegamos la base de datos en el panel izquierdo, vamos al apartado de *Tables* y hacemos clic en el botón +. En la parte derecha completamos el *script* de creación de la tabla con los campos, tipos y restricciones que queramos. Finalmente, pulsamos el botón *play* de la esquina superior derecha.




Finalmente, si queremos guardar los datos en un archivo *script* de *backup*, hacemos clic derecho sobre la base de datos, y elegimos la opción *Export Data* (si queremos guardar las tablas con sus datos) o *Export Struct* (si sólo nos interesa la estructura de las tablas, sin la información que contienen).




# 2. La librería *mysql2*

Conviene tener presente que la combinación de Node.js y MySQL no es demasiado habitual. Es bastante más frecuente el uso de bases de datos MongoDB empleando este framework, ya que la información que se maneja, en formato BSON, es muy fácilmente exportable entre los dos extremos.

Sin embargo, también existen herramientas para poder trabajar con MySQL desde Node.js. Una de las más populares es la librería `mysql2`, disponible en el repositorio NPM.

> **NOTA**: existe una versión anterior de la librería, llamada *mysql*. Sin embargo, el mecanismo de autenticación de las últimas versiones de MySQL Server no es compatible con esta librería, por lo que es

> más recomendable emplear esta última versión.

Para ver cómo utilizarla, comenzaremos por crear un proyecto llamado "*PruebaContactosMySQL*" en nuestra carpeta de "*ProyectosNode/Pruebas*". Crea dentro un archivo `index.js`, y ejecuta el comando `npm init` para inicializar el archivo `package.json`. Después, instalamos la librería con el correspondiente comando `npm install`:

```
npm install mysql2
```

## 2.1. Conexión a la base de datos

Una vez instalado el módulo, en nuestro archivo `index.js` lo importamos (con `require`), y ejecutamos el método `createConnection` para establecer una conexión con la base de datos, de acuerdo a los parámetros de conexión que facilitaremos en el propio método:

- `host`: nombre del servidor (normalmente `localhost`)
- `user`: nombre del usuario para conectar
- `password`: password del usuario para conectar
- `database`: nombre de la base de datos a la que acceder, de entre las que haya disponibles en el servidor al que conectamos.
- `port`: un parámetro opcional, a especificar si el servidor de bases de datos está escuchando por un puerto que no es el puerto por defecto
- `charset`: también opcional, para indicar un juego de codificación de caracteres determinado (por ejemplo, "utf8").

En el caso de la máquina virtual del curso, disponemos del usuario *alumno* con contraseña *alumno1234*. Si utilizáis otro sistema diferente, como XAMPP, normalmente el usuario predefinido es *root* con la contraseña vacía.

Para las pruebas que haremos en este proyecto de prueba, utilizaremos una base de datos llamada "*contactos*" que puedes descargar, descomprimir e importar desde aquí. Teniendo en cuenta todo lo anterior, podemos dejar los parámetros de conexión así:

```
const mysql = require('mysql2');

let conexion = mysql.createConnection({
    host: "localhost",
    user: "alumno",
    password: "alumno1234",
    database: "contactos"
});
```

Después, podemos establecer la conexión con:

```
conexion.connect((error) => {
    if (error)
        console.log("Error al conectar con la BD:", err);
    else
        console.log("Conexión satisfactoria");
});
```

En el caso de que se produzca algún error de conexión, lo identificaremos en el parámetro `error` y podremos actuar en consecuencia. En este caso se muestra un simple mensaje por la consola, pero también podemos almacenarlo en algún *flag* booleano o algo similar para impedir que se hagan operaciones contra la base de datos, o se redirija a otra página.

## 2.2. Consultas

La base de datos "contactos" tiene una tabla del mismo nombre, con los atributos *id*, *nombre* y *telefono*.

| id | nombre | telefono |
|---|---|---|
| 1 | Nacho Iborra | 966112233 |
| 2 | Arturo Bernal | 965665544 |
| 3 | Alex Amat | 966998877 |

Vamos a definir una consulta para obtener resultados y recorrerlos. Por ejemplo, mostrar todos los contactos:

```
conexion.query("SELECT * FROM contactos",
(error, resultado, campos) => {
    if (error)
        console.log("Error al procesar la consulta");
    else
    {
        resultado.forEach((contacto) => {
            console.log(contacto.nombre, ":",
                contacto.telefono);
        });
    }
});
```

Notar que el método `query` tiene dos parámetros: la consulta a realizar, y un *callback* que recibe otros tres parámetros: el error producido (si lo hay), el conjunto de resultados (que se puede procesar como un vector de objetos), e información adicional sobre los campos de la consulta.

Notar también que el propio método `query` nos sirve para conectar (dispone de su propio control de error), por lo que no sería necesario el paso previo del método `connect`. En cualquier caso, podemos

hacerlo si queremos asegurarnos de que hay conexión, pero cada *query* que hagamos también lo puede verificar.

Existen otras formas de hacer consultas:

- Utilizando marcadores (*placeholders*) en la propia consulta. Estos marcadores se representan con el símbolo `?`, y se sustituyen después por el elemento correspondiente de un vector de parámetros que se coloca en segunda posición. Por ejemplo:

```
conexion.query("SELECT * FROM contactos WHERE id = ?", [1],
    (error, resultado, campos) => {
        ...
```

- Utilizando como parámetro del método `query` un objeto con diferentes propiedades de la consulta: la instrucción SQL en sí, los parámetros embebidos mediante *placeholders*... de forma que podemos proporcionar información adicional como *timeout*, conversión de tipos, etc.

```
conexion.query({
    sql: "SELECT * FROM contactos WHERE id = ?",
    values: [1],
    timeout: 4000
}, (error, resultado, campos) => {
    ...
```

## 2.3. Actualizaciones (inserciones, borrados, modificaciones)

Si lo que queremos es realizar alguna modificación sobre los contenidos de la base de datos (INSERT, UPDATE o DELETE), estas operaciones se realizan desde el mismo método `query` visto antes. La diferencia está en que en el parámetro `resultado` del callback ya no están los registros de la consulta, sino datos como el número de filas afectadas (en el atributo `affectedRows`), o el *id* del nuevo elemento insertado (atributo `insertId`), en el caso de inserciones con id autonumérico.

Por ejemplo, si queremos insertar un nuevo contacto en la agenda y obtener el *id* que se le ha asignado, lo podemos hacer así:

```
conexion.query("INSERT INTO contactos" +
"(nombre, telefono) VALUES " +
"('Fernando', '966566556')", (error, resultado, campos) => {
    if (error)
        console.log("Error al procesar la inserción");
    else
        console.log("Nuevo id = ", resultado.insertId);
});
```

También podemos pasar un objeto JavaScript como dato a la consulta, y automáticamente se asigna cada campo del objeto al campo correspondiente de la base de datos (siempre que los nombres de los campos coincidan). Esto puede emplearse tanto en inserciones como en modificaciones:

```
conexion.query("INSERT INTO contactos SET ?",
    {nombre: 'Nacho C.', telefono: '965771111'},
    (error, resultado, campos) => {
        ...
});
```

Si hacemos un borrado o actualización, podemos obtener el número de filas afectadas, de esta forma:

```
conexion.query("DELETE FROM contactos WHERE id > 10",
    (error, resultado, campos) => {
        if (error)
            console.log("Error al realizar el borrado");
        else
            console.log(resultado.affectedRows,
                "filas afectadas");
});
```