Acceso a MySQL desde Node

En este documento veremos cómo acceder a bases de datos relacionales (en concreto, sistemas MySQL) desde Node.js. Explicaremos qué módulo vamos a utilizar para ello, y cómo realizar con él las operaciones básicas: conectar a la base de datos, operaciones de inserción/borrado/modificación y consultas o listados.

1. 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 "*ContactosMySQL*" en nuestra carpeta de "*ProyectosNode/Pruebas*". Dentro 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. 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").

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, aunque ya lo hemos pedido en un ejercicio previo. Teniendo en cuenta todo lo anterior, podemos dejar los parámetros de conexión así:

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

// Cambiar usuario y contraseña por los que tengamos en
// nuestro sistema
let conexion = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "root",
    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.

3. 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) => {
    ...
```

4. 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");
});
```

> **Ejercicio 1:**

> Crea una carpeta llamada "**LibrosMySQL**" dentro de la carpeta "*ProyectosNode/Ejercicios*". Crea una base de datos llamada *libros* en MySQL e importa el *backup* que tienes disponible comprimido aquí.
>
> En el proyecto, inicializa el archivo `package.json` con `npm init`, e instala el módulo *mysql2*. Crea un archivo `index.js` que cargue este módulo (con `require`), y realice estas operaciones sobre la base de datos de libros:
>
> - Insertar 3 libros cualesquiera, con los datos que se te ocurran (observa qué campos tiene la tabla).
> - Listar los libros de más de 10 euros.
> - Modificar el precio del libro 1 a 35 euros.
> - Borrar el libro 3.