Consultas (práctica)

Tiempo estimado: 15min

El objeto de esta práctica es afianzar, reforzar y consolidar los conocimientos teóricos presentados en la lección.

Al finalizarla, el estudiante:

- Habrá creado una base de datos SQLite.
- Habrá abierto una conexión a una base de datos SQLite.
- Habrá consultado una base de datos SQLite.
- Habrá trabajado con transacciones.

Objetivos

El objetivo de la práctica es divertirnos con el paquete sqlite3. Para ello, vamos a crear una base de datos SQLite y vamos a ejecutar consultas y transacciones contra ella.

Preparación del entorno

Recordemos que no se recomienda utilizar una versión instalada globalmente del *driver*, a menos que sea necesario. Por buenas prácticas, se realiza una instalación local al proyecto. Esto consiste en añadir el *driver* a las dependencias del archivo package.json.

En la práctica, simplemente instalaremos el *driver* localmente mediante npm:

1. Abrir una consola.
2. Crear el directorio de la práctica e ir a él.
3. Instalar el *driver* localmente:

   ```
   $ npm install sqlite3
   ```
   npm creará el directorio node_modules donde se instalará el *driver*.
4. Abrir node en modo interactivo.
5. Importar el *driver*:

   ```
   > const sqlite = require("sqlite3")
   undefined
   > sqlite
   { Database: [Function: Database],
     Statement: [Function: Statement],
     OPEN_READONLY: 1,
     OPEN_READWRITE: 2,
     OPEN_CREATE: 4,
     VERSION: '3.15.0',
     SOURCE_ID: '2016-10-14 10:20:30 707875582fcba352b4906a595ad89198d84711d8',
     VERSION_NUMBER: 3015000,
     OK: 0,
     ERROR: 1,
     INTERNAL: 2,
     PERM: 3,
     ABORT: 4,
     BUSY: 5,
     LOCKED: 6,
     NOMEM: 7,
     READONLY: 8,
     INTERRUPT: 9,
     IOERR: 10,
     CORRUPT: 11,
   ```



```
  NOTFOUND: 12,
  FULL: 13,
  CANTOPEN: 14,
  PROTOCOL: 15,
  EMPTY: 16,
  SCHEMA: 17,
  TOOBIG: 18,
  CONSTRAINT: 19,
  MISMATCH: 20,
  MISUSE: 21,
  NOLFS: 22,
  AUTH: 23,
  FORMAT: 24,
  RANGE: 25,
  NOTADB: 26,
  cached: { Database: [Function: Database], objects: {} },
  verbose: [Function] }
>
```

Conexión a base de datos

Ahora, vamos a abrir una conexión a la base de datos. Recordemos que para ello se usa la clase Database del paquete sqlite3. En el momento de abrir la conexión, podemos indicar cómo debe actuar el *driver* si no existe el archivo de datos: crearlo o fallar en la apertura. También podemos indicar si deseamos abrir la conexión en modo lectura o lectura/escritura. Ambas cosas se indican mediante el parámetro mode.

1. Ir a la consola.
2. Abrir una conexión a la base de datos mibd.db, creándola si no existe, que es nuestro caso:

```
> db = new sqlite.Database("./mibd.db", sqlite.OPEN_READWRITE|sqlite.OPEN_CREATE,
function(err) { if (err) console.error(err); })
Database { open: false, filename: './mibd.db', mode: 6 }
>
```

3. Mostrar las propiedades enumerables de la conexión:

```
> db
Database { open: true, filename: './mibd.db', mode: 6 }
>
```

4. Listar las entradas del directorio actual:

```
> fs.readdirSync(".")
[ 'mibd.db', 'node_modules' ]
>
```

Como se observa, el *driver* ha creado el archivo de datos.

Consultas simples

A continuación, vamos a ejecutar algunas consultas:

1. Ir a la consola.
2. Crear la tabla Tabla:

```
> db.run("CREATE TABLE Tabla(x, y, z)", function(err) { if (err)
console.error(err); else console.log("Tabla creada."); })
Database { open: true, filename: './mibd.db', mode: 6 }
> Tabla creada.
```

3. Insertar varias filas de datos:

```
> db.run("INSERT INTO Tabla VALUES(1, 2, 3)", function(err) { if (err)
console.error(err); else console.log("Fila insertada."); })
Database { open: true, filename: './mibd.db', mode: 6 }
> Fila insertada.
> db.run("INSERT INTO Tabla VALUES(3, 2, 1)", function(err) { if (err)
console.error(err); else console.log("Fila insertada."); })
Database { open: true, filename: './mibd.db', mode: 6 }
> Fila insertada.
```

4. Consultar las filas almacenadas en la tabla de prueba mediante el método all():


```
> db.all("SELECT * FROM Tabla", function(err, rows) { if (err) console.error(err);
else console.dir(rows); })
Database { open: true, filename: './mibd.db', mode: 6 }
> [ { x: 1, y: 2, z: 3 }, { x: 3, y: 2, z: 1 } ]
```

5. Consultar las filas almacenadas en la tabla de prueba mediante el método each():

```
> db.each("SELECT * FROM Tabla", function(err, row) { if (err) console.error(err);
else console.dir(row); })
Database { open: true, filename: './mibd.db', mode: 6 }
> { x: 1, y: 2, z: 3 }
{ x: 3, y: 2, z: 1 }
```

Recuerde que el método each() ejecuta la función *callback* con cada fila. No como all() que lo hace para el conjunto resultado completo.

6. Consultar las filas almacenadas en la tabla NoExiste:

```
> db.all("SELECT * FROM NoExiste", function(err, rows) { if (err)
console.error(err); else console.dir(rows); })
Database { open: true, filename: './mibd.db', mode: 6 }
> { Error: SQLITE_ERROR: no such table: NoExiste errno: 1, code: 'SQLITE_ERROR' }
```

Consultas preparadas

Veamos cómo trabajar con una sentencia que sabemos invocaremos varias veces con distintos valores:

1. Ir a la consola.
2. Crear la sentencia preparada:

```
> cmd = db.prepare("INSERT INTO Tabla VALUES($x, $y, $z)", function(err) { if
(err) console.error(err); })
Statement { sql: 'INSERT INTO Tabla VALUES($x, $y, $z)' }
>
```

3. Usar la sentencia preparada para insertar dos nuevas filas:

```
> cmd.run({$x: 123, $y: 456, $z: 789}, function(err) { if (err)
console.error(err); })
Statement { sql: 'INSERT INTO Tabla VALUES($x, $y, $z)' }
> cmd.run({$x: 321, $y: 654, $z: 987}, function(err) { if (err)
console.error(err); })
Statement {
  sql: 'INSERT INTO Tabla VALUES($x, $y, $z)',
  lastID: 3,
  changes: 1 }
>
```

4. Consultar el contenido de la tabla:

```
> db.all("SELECT * FROM Tabla", function(err, rows) { if (err) console.error(err);
else console.dir(rows); })
Database { open: true, filename: './mibd.db', mode: 6 }
> [ { x: 1, y: 2, z: 3 },
  { x: 3, y: 2, z: 1 },
  { x: 123, y: 456, z: 789 },
  { x: 321, y: 654, z: 987 } ]
```

Transacciones explícitas

Recordemos que el *driver* trabaja de manera predeterminada en modo autoconfirmación, lo que significa que tras cada comando SQL ejecutado, hay un COMMIT o ROLLBACK implícito, según el resultado de la ejecución. Ahora, vamos a trabajar con transacciones explícitas:

1. Ir a la consola.
2. Insertar dos nuevas filas mediante una transacción:

```
> db.run("BEGIN", function(err) {
... if (err) return console.error(err);
... db.run("INSERT INTO Tabla VALUES(?, ?, ?)", [135, 246, 789], function(err) {
..... if (err) return console.error(err);
..... db.run("INSERT INTO Tabla VALUES(?, ?, ?)", [531, 642, 987], function(err) {
....... if (err) return console.error(err);
```



```
..... db.run("COMMIT", function(err) {
......... if (err) return console.error(err);
......... });
..... });
..... });
... });
Database { open: true, filename: './mibd.db', mode: 6 }
> db.all("SELECT * FROM Tabla", function(err, rows) { if (err) console.error(err);
else console.dir(rows); })
Database { open: true, filename: './mibd.db', mode: 6 }
> [ { x: 1, y: 2, z: 3 },
  { x: 3, y: 2, z: 1 },
  { x: 123, y: 456, z: 789 },
  { x: 321, y: 654, z: 987 },
  { x: 135, y: 246, z: 789 },
  { x: 531, y: 642, z: 987 } ]
```

