Consultas

Tiempo estimado: 25min

Una vez tenemos claro cuál es el paquete que implementa el *driver* de SQLite a utilizar, lo siguiente natural es aprender a conectar a bases de datos SQLite y realizar consultas. Veamos cómo hacerlo.

Comenzamos la lección describiendo el proceso de conexión a una base de datos SQLite. Una vez sabemos cómo conectar, a continuación, presentamos las distintas formas de ejecutar consultas SQL. Y finalmente, cómo usar transacciones.

Al finalizar la lección, el estudiante sabrá:

- Qué es una conexión.
- Cómo abrir y cerrar conexiones.
- Cómo ejecutar comandos SQL.
- Cómo usar transacciones.

Conexiones

Una conexión (*connection*) es un objeto que representa una sesión a una base de datos. Es el enlace a través del cual se ejecutan los comandos SQL contra la instancia de base de datos.

En SQLite, la conexiones se representan mediante instancias de la clase Database. Recordemos que, en toda conexión a una base de datos SQLite, hay una base de datos principal y cero, una o más bases de datos secundarias. La base de datos, con la que se abre la conexión a través de la clase Database, se considera la base de datos principal. Cualquier otra base de datos abierta, a través de la conexión usando el comando ATTACH DATABASE, se considera secundaria.

Apertura de conexión

Cada vez que creamos una instancia de la clase Database, estamos creando y abriendo la sesión a la base de datos. He aquí las sobrecargas del constructor de la clase:

```
constructor(filename)
constructor(filename, mode)
constructor(filename, callback)
constructor(filename, mode, callback)
```

Parámetro	Tipo de datos	Descripción
filename	String	Ruta al archivo de datos de la base de datos SQLite; una cadena vacía para una base de datos anónima en disco; o :memory: para una base de datos en memoria.
mode	Number	Modo de apertura de la base de datos: • sqlite3.OPEN_READONLY: Conexión de sólo lectura. • sqlite3.OPEN_READWRITE: Conexión de L/E. • sqlite3.OPEN_CREATE: Crear la base de datos si no existe. Valor predeterminado: OPEN_READWRITE | OPEN_CREATE. Abre la base de datos para L/E y la crea si no existe.
callback	Function	Función a invocar cuando se ha abierto la conexión con la base de datos: fn(error).

Ejemplo:

```
db = new sqlite.Database("test.db", function(err) {
  if (err) console.error(err);
});
```


Propiedades de conexión

La clase Database dispone de varias propiedades que permiten conocer algunas propiedades de la conexión:

Propiedad	Tipo de datos	Descripción
filename	String	Archivo de datos de la base de datos principal de la conexión. Si la base de datos principal es una base de datos en memoria, contendrá :memory:.
open	Boolean	¿Conexión abierta?
mode	Integer	Modo de apertura de la conexión.

Eventos de conexión

La clase Database es una clase emisora de eventos, al igual que muchas que podemos encontrar en Node. Esta clase proporciona los siguientes eventos:

Evento	Controlador	Descripción
error	fn(error)	Se genera cuando se produce un error.
open	fn()	Se produce cuando se abre la conexión con la base de datos.
close	fn()	Se dispara cuando se cierra la conexión con la base de datos.
trace	fn(sql)	Se genera cada vez que se va a ejecutar una consulta SQL contra la base de datos. El parámetro sql contiene el texto de la sentencia a ejecutar.
profile	fn(sql, time)	Se genera cada vez que se finaliza la ejecución de una consulta. El parámetro sql contiene el texto de la consulta ejecutada. Y time, el tiempo aproximado de ejecución.

Recordemos que, en Node, los emisores de eventos permiten registrar controladores de eventos mediante su método on(). Ejemplo:

```
db.on("trace", function(sql) {
  console.log("Se va a ejecutar la siguiente consulta:", sql);
});
```

Configuración de conexión

El *driver* permite configurar varios aspectos de conexión relacionados con la supervisión de los comandos ejecutados y el tiempo máximo que se invierte en la ejecución de un comando SQL. Para ello, se utiliza el método configure() de la clase Database:

```
configure(option, value)
```

Parámetro	Tipo de datos	Descripción
busyTimeout	Number	Tiempo de ocupación máximo.
trace	function	Función a invocar cada vez que se va a ejecutar una consulta SQL: fn(sql).
profile	function	Función a invocar cada vez que finaliza la ejecución de una consulta SQL: fn(sql, time).

Ejemplo:

```
db.configure("busyTimeout", 123456);
```

Cierre de conexión

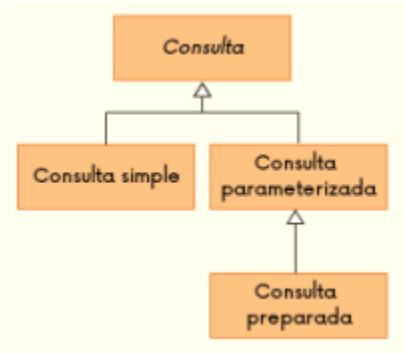
Por buenas prácticas, se recomienda el cierre explícito de una conexión cuando ya no se va a utilizar más para, así, liberar sus recursos y cualquier posible bloqueo que pudiera tener adquirido en la base de datos. Para ello, se utiliza el método close() de la clase Database:

```
close()
close(callback)
```


Parámetro	Tipo de datos	Descripción
callback	Function	Función a invocar cuando la conexión se ha cerrado: fn(error).

Consultas

Una vez se dispone de una conexión con la base de datos, lo siguiente es ejecutar sentencias SQL contra ella. Esto se puede hacer de distintas formas, mediante sentencias simples, sentencias parametrizadas o sentencias preparadas.



Consultas simples

Una sentencia simple (*simple statement*), también conocida como sentencia regular (*standard statement*), es aquella que proporciona una sentencia SQL literal. Tal cual se redacta en la aplicación, se envía al motor de consultas de SQLite.

Para la ejecución de este tipo de sentencias, la clase Database proporciona los métodos run(), each(), all() y get().

Método Database.run()

El método run() se utiliza para ejecutar una sentencia SQL contra la base de datos que no espera un conjunto resultado, generalmente, comandos DDL o DML:

```
run(sql) : Database
run(sql, callback) : Database
```

Parámetro	Tipo de datos	Descripción
sql	String	Consulta a ejecutar.
callback	Function	Función a ejecutar al finalizar su ejecución: fn(error).

El método devuelve la instancia Database por si deseamos encadenar varias consultas una detrás de otra. Esto también es así con los demás métodos de consulta.

A continuación, se muestra un ejemplo de cómo ejecutar una instrucción DDL:

```
db.run("CREATE TABLE T(x, y)", function(err) {
  if (err) console.error(err);
});
```

Método Database.each()

El método each() se utiliza para la ejecución de comandos que devuelven un conjunto de filas como resultado, por ejemplo, para la ejecución de SELECT y/o alguna directiva PRAGMA:

```
each(sql, callback) : Database
each(sql, callback, complete) : Database
```

Parámetro	Tipo de datos	Descripción
sql	String	Consulta a ejecutar.
callback	Function	Función a ejecutar con cada fila de datos: fn(error, row).
complete	Function	Función a ejecutar una vez se ha recorrido todas las filas

del resultado: fn(error, count).
El parámetro count indica el número de filas afectadas por la sentencia SQL ejecutada.

Ejemplo:

```
db.each("SELECT * FROM T", function(err, row) {
  if (err) {
    console.error(err);
  } else {
    console.log("x:", row.x, "; y:", row.y);
  }
});
```

Método Database.all()

El método all() es similar a each(), pero devuelve todo el conjunto resultado de golpe:

```
all(sql, callback) : Database
```

Parámetro	Tipo de datos	Descripción
sql	String	Consulta a ejecutar.
callback	Function	Función a ejecutar cuando se dispone del resultado: fn(error, rows). Donde rows es un *array* con las filas que forman el resultado.

He aquí un ejemplo ilustrativo:

```
db.all("SELECT * FROM T", function(err, rows) {
  if (err) {
    console.error(err);
  } else {
    for (let row of rows) {
      console.log("x:", row.x, "; y:", row.y);
    }
  }
});
```

Método Database.get()

El método get() ejecuta la sentencia SQL y permite el acceso únicamente a la primera fila del resultado:

```
get(sql, callback) : Database
```

Parámetro	Tipo de datos	Descripción
sql	String	Consulta a ejecutar.
callback	Function	Función a ejecutar con la primera fila del resultado: fn(error, row).

Veamos un ejemplo:

```
db.get("SELECT * FROM T", function(err, row) {
  if (err) {
    console.error(err);
  } else {
    console.log("x:", row.x, "; y:", row.y);
  }
});
```

Consultas parametrizadas

Una sentencia parametrizada (*parameterized statement*) es un tipo especial de consulta que contiene parámetros en el texto del comando SQL a ejecutar, cuyos valores se deben fijar antes de la ejecución de la consulta. He aquí un ejemplo de sentencia parametrizada:

```
SELECT * FROM T WHERE x = ?
```

Los distintos parámetros se indican en la consulta mediante marcadores.

Ejecución de sentencia parametrizadas

Para la ejecución de sentencias parametrizadas, se utiliza los mismos métodos que con las sentencias simples, pero usando el parámetro params para indicar la lista de parámetros, ya sea un *array* o un objeto, según el tipo de marcador utilizado:

```
run(sql, params) : Database
run(sql, params, callback) : Database
each(sql, params, callback) : Database
each(sql, params, callback, complete) : Database
all(sql, params, callback) : Database
get(sql, params, callback) : Database
```

Marcadores de parámetros

Un marcador de parámetro (*parameter marker*) es una especie de variable que identifica un valor que debe proporcionarse explícitamente en el momento de ejecutar la sentencia, pudiéndose reutilizar la misma sentencia con distintos parámetros.

En SQLite, se puede utilizar dos tipos de parámetros, los nombrados y los posicionales.

Un marcador posicional (*positional parameter*) es aquel que se indica en el texto de la consulta mediante un signo de interrogación (?). Como una consulta puede disponer de varios de estos marcadores, el mapeo de los valores a los marcadores se realiza posicionalmente, o sea, el parámetro 0 corresponde a la primera aparición del marcador, el parámetro 1 al segundo y así sucesivamente.

Veámoslo mediante un ejemplo ilustrativo:

```
db.all("SELECT * FROM T WHERE x = ? or y = ?", [1,4], function(err, rows) {
  if (err) {
    console.error(err);
  } else {
    for (let row of rows) {
      console.log("x:", row.x, "; y:", row.y);
    }
  }
});
```

En el ejemplo anterior, los parámetros se mapearán a la siguiente consulta:

```
SELECT * FROM T WHERE x = 1 or y = 4
```

En cambio, un marcador con nombre (*named marker*) es aquel que se representa mediante un nombre identificativo, haciendo así más fácil su utilización. Ahora, en vez de pasar un *array* con los parámetros, se pasa un objeto, donde cada propiedad corresponde a un parámetro. Cada marcador puede seguir uno de los siguientes formatos:

```
$marcador
@marcador
:marcador
```

Veamos el ejemplo anterior, pero con marcadores con nombre:

```
db.all("SELECT * FROM T WHERE x = $x or y = $y", {$x: 1, $y: 4}, function(err, rows) {
  if (err) {
    console.error(err);
  } else {
    for (let row of rows) {
      console.log("x:", row.x, "; y:", row.y);
    }
  }
});
```


Consultas preparadas

Una sentencia preparada (*prepared statement*) es una sentencia parametrizada con un plan de ejecución cacheado, siendo este tipo de sentencia de ejecución más rápido porque no es necesario recompilarlas con cada ejecución. El funcionamiento es como sigue:

1. Preparar la sentencia.

   Esto se hace una única vez por sentencia preparada.

2. Ejecutar la sentencia.

   Esto se hace en dos pasos. El primero proporciona los parámetros. Y el segundo ejecuta la sentencia con los parámetros del último mapeo.

Preparación de la sentencia

Lo primero que hay que hacer es obtener una instancia de la clase Statement, que representa una sentencia preparada. Una vez tengamos la sentencia, utilizaremos sus métodos asociados para ejecutar la sentencia con distintos parámetros.

Para obtener una instancia Statement, hay que utilizar el método prepare() de la clase Database:

```
prepare(sql) : Statement
prepare(sql, params) : Statement
prepare(sql, callback) : Statement
prepare(sql, params, callback) : Statement
```

Parámetro	Tipo de datos	Descripción
sql	String	Consulta a preparar.
params	Array u Object	Parámetros iniciales de la sentencia preparada.
callback	Function	Función a ejecutar tras haberse preparado la sentencia: fn(error).

A continuación, se ilustra cómo crear una sentencia preparada de ejemplo:

```
var stmt = db.prepare("DELETE FROM T WHERE x = ? or y = ?", function(err) {
  if (err) {
    console.log(err);
  }
});
```

Mapeo de parámetros

Una vez se dispone de la sentencia preparada, lo siguiente es proporcionar los parámetros para la siguiente ejecución. Este proceso hay que hacerlo con cada ejecución. Para ello, se utiliza su método bind():

```
bind(params) : Statement
bind(params, callback) : Statement
```

Parámetro	Tipo de datos	Descripción
params	Array u Object	Valores de los marcadores de la sentencia.
callback	Function	Función a ejecutar tras haberse fijado los valores a los marcadores de la sentencia: fn(error).

El método devuelve la propia instancia Statemente por si deseamos encadenar varias invocaciones de método.

Ejemplos:

```
stmt.bind([1, 4]);
stmt.bind({$x: 123, $y: 321});
```

Ejecución de la sentencia

Una vez se ha fijado los parámetros con los que ejecutar la sentencia, se procede a su ejecución, mediante los métodos run(), each(), all() o get() de la sentencia:

```
run() : Statement
```



```
run(callback) : Statement
each(callback) : Statement
each(callback, complete) : Statement
all(callback) : Statement
get(callback) : Statement
```

Ejemplo ilustrativo:

```
stmt.all(function(err, rows) {
  if (err) {
    console.error(err);
  } else {
    for (let row of rows) {
      console.log("x:", row.x, "; y:", row.y);
    }
  }
});
```

Es posible ejecutar la sentencia preparada pasando los parámetros en el método ejecutor, es decir, omitiendo el método bind() y pasando los valores de los marcadores en el propio método de ejecución:

```
run(params) : Statement
run(params, callback) : Statement
each(params, callback) : Statement
each(params, callback, complete) : Statement
all(params, callback) : Statement
get(params, callback) : Statement
```

Veamos un ejemplo completo:

```
//(1) preparamos
var stmt = db.prepare("SELECT * FROM T WHERE x = ? or y = ?");

//(2) ejecutamos
stmt.all([1, 4], function(err, rows) {
  if (err) {
    console.error(err);
  } else {
    for (var row of rows) {
      console.log("x:", row.x, "; y:", row.y);
    }
  }
});

stmt.all([2, 8], function(err, rows) {
  if (err) {
    console.error(err);
  } else {
    for (var row of rows) {
      console.log("x:", row.x, "; y:", row.y);
    }
  }
});
```

Finalización de ejecución

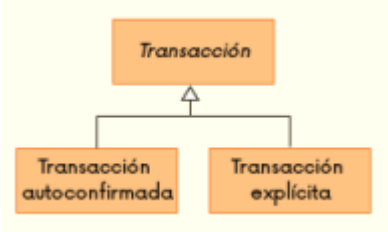
Cuando se sabe que una sentencia preparada ya no se va a utilizar, se recomienda cerrar la sentencia para liberar los recursos que tiene asignados. Para ello, se utiliza el método finalize():

```
finalize()
finalize(callback)
```

Parámetro	Tipo de datos	Descripción
callback	Function	Función a ejecutar tras haberse cerrado la sentencia: fn(error).

Transacciones

A diferencia de algunos *drivers* que proporcionan métodos especiales a nivel de conexión para dar soporte a transacciones como, por ejemplo, setAutocommit(), el *driver* sqlite3 no proporciona ninguno, pero sigue permitiendo las transacciones. Recordemos los dos tipos de transacciones soportadas por SQLite, las autoconfirmadas y las explícitas.



Transacciones autoconfirmadas

De manera predeterminada, SQLite funciona bajo modo autoconfirmación (*autocommit mode*). Cada vez que se ejecuta un comando SQL, el motor de base de datos ejecuta un COMMIT implícito si el comando finaliza correctamente; en otro caso, realiza un ROLLBACK para deshacer cualquier cambio que haya realizado antes del fallo.

Recordemos que, si estamos usando transacciones autoconfirmadas, no es posible utilizar los comandos de finalización de transacción COMMIT y ROLLBACK. Sólo se pueden utilizar si estamos usando una transacción explícita.

Transacciones explícitas

Si la transacción la forman varias instrucciones SQL, es necesario delimitarlas mediante los comandos BEGIN y COMMIT o ROLLBACK, es decir, hay que usar transacciones explícitas (*explicit transactions*).

A continuación, un ejemplo ilustrativo:

```
db.run("BEGIN", function(err) {
  db.run("INSERT INTO T VALUES(5, 6)", function(err) {
    if (err) {
      console.error(err);
    } else {
      db.run("INSERT INTO T VALUES(7, 8)", function(err) {
        if (err) {
          console.error(err);
        } else {
          db.run("COMMIT", function(err) {
            if (err) {
              console.error(err);
            }
          });
        }
      });
    }
  });
});
```

Cuando se produce un error durante la transacción, por ejemplo, en un comando una vez se ha iniciado correctamente mediante BEGIN, es importante recordar que la transacción no se cierra automáticamente, es decir, no hay un ROLLBACK implícito, como es de esperar. Si es necesario, debemos ejecutarlo nosotros mismos. Si no lo hacemos, e intentamos abrir otra transacción, el sistema propagará un error indicando que ya hay una transacción activa:

```
SQLITE_ERROR: cannot start a transaction within a transaction errno: 1
```

