SQLite for Node.js

Introduction to SQLite

SQLite is a **lightweight, serverless, and self-contained** database engine widely used in applications, browsers, and mobile devices.

What Is SQLite?

- A **relational database** system (RDBMS)
- Stores data in a **single file** (`.db` or `.sqlite`)
- Requires **no server process** — it runs directly inside the application
- Uses **SQL (Structured Query Language)** to manage data

SQLite Everywhere!

SQLite is embedded inside countless applications, systems, and devices — not something you install, but something that's already there.

- **Web browsers:** Chrome, Safari, Firefox, Edge — all use SQLite internally
- **Operating systems:** macOS, iOS, Android, Windows 10/11
- **Phones, TVs, and IoT devices:** almost every smartphone, smart TV, and embedded system stores settings or logs in SQLite
- **Applications:** Dropbox, Skype, Zoom, Adobe, Spotify, WhatsApp, and many more

Why Use SQLite?

Advantage	Description
Simple	No setup or configuration — just use the `.db` file
Portable	Works on all platforms (macOS, Windows, Linux, Android, iOS)
Fast	Lightweight engine optimized for embedded use
Reliable	ACID-compliant — supports atomic transactions

Basic Structure

Concept	Description	Example
Database	Single file storing all data	`data.db`
Table	Organized rows & columns	`CREATE TABLE users (...);`
Row	A single record	`(1, "Alice", "2025-11-07")`
Column	Field of data	`id`, `name`, `date`

Common Commands (CRUD Operations)

```
CREATE TABLE users (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  age INTEGER
);

INSERT INTO users (name, age) VALUES ('Alice', 25);

SELECT * FROM users;

UPDATE users SET age = 26 WHERE name = 'Alice';

DELETE FROM users WHERE id = 1;
```

How to Use in Node.js

Package installation.

```
npm install sqlite3
import sqlite3 from 'sqlite3';
```

Open Database

```
const db = new sqlite3.Database('data.db');
```

With error checking:

```
const db = new sqlite3.Database('data.db', (err) => {
  if (err) {
    console.error('Failed to open DB:', err.message);
    process.exit(1);
  }
});
```

Usage

In node.js, we make a SQL string and use it the argument of functions.

```
const sql = 'SELECT * FROM posts WHERE title LIKE ?';
db.all(sql, ['%hello%'], (err, rows) => {
  if (err) return console.error(err.message);
  console.log(rows);
});
```

Output example:

```
[
  { _id: 1, title: 'hello world', date: '2025-11-04' },
  {_id: 2, title: 'say hello again', date: '2025-11-05' }
]
```

Exec, Run, Get, and All functions

Function	Returns	Used For
`db.exec()`	No rows	Running multiple setup statements
`db.run()`	No rows (but has `lastID`, `changes`)	Insert, update, or delete data
`db.get()`	Single row (first match only)	Fetch one record
`db.all()`	Array of rows	Fetch multiple records at once
`db.each()`	Calls callback once per row	Stream through rows one by one

Exec for Running Setup Statements

```
db.exec(`
  PRAGMA foreign_keys = ON;
  CREATE TABLE IF NOT EXISTS posts (...);
`, (err) => { ... });
```

You can run multiple SQL commands in one call — good for setup.

- The lambda `(err) => { ... }` that you pass as an argument is executed **after** the SQL commands complete.

Run for CUD (not for Read)

```
db.run(
  'INSERT INTO posts (title, date) VALUES (?, ?)',
  [title, date],
  function(err) { console.log(this.lastID); }
);
```

Used for a single SQL operation that modifies data and might need results like the inserted row ID.

- The array [title, date] is the parameter value used to safely fill in the ? placeholder in your SQL query.

Get for Single Read and All for Reading All

```
db.get(sql, [params], callback)
db.all(sql, [params], callback)
```

Runs a SELECT query and returns all matching rows at once as an array.

Using callback for get vs all

```
// One record
db.get('SELECT * FROM users WHERE id = ?', [id], (err, row) => {
  if (row) console.log(row);
});

// All records
db.all('SELECT * FROM users', [], (err, rows) => {
  rows.forEach(r => console.log(r));
});
```

- db.get() → "Run a SELECT query and give me just one row, and give me the next one with the get()"
- db.all() → "Run a SELECT query and give me all rows at once, and I'll process them on my own."

`this` in SQLite

In this code, we use `this`.

```
db.run(sql, [title, date, id], function (err2) {
  if (err2) { ... }
  if (this.changes === 0) { ... }   // ← here!
  ...
});
```

- In Node.js's sqlite3 library, the callback you pass to db.run() is called after the SQL statement finishes running.
- When that callback runs, the library sets `this` to the Statement object that represents that SQL command.
- This is why we should not use the arrow function (=>) that uses `this` in the lexical scope.

Inside the Statement object

The Statement object contains useful metadata about what just happened:

Property Description Example:

- `this.lastID` : The ID of the last inserted row (for INSERT) e.g., 5
- `this.changes` : The number of rows affected (for UPDATE or DELETE) e.g., 1

So this code checks whether the UPDATE changed any rows at all.

```
if (this.changes === 0) { console.log('null'); }
```

Real Example: index.js

The code can be found in `code/sqlite` directory.

Open DB

```
const path = require('path');

const DB_PATH = path.join(__dirname, 'todo.sqlite');
const db = new sqlite3.Database(DB_PATH, (err) => {
  if (err) {
    console.error('Failed to open DB:', err.message);
    process.exit(1);
  }
});
```

SQLite automatically creates the database file if it doesn’t already exist.

Close DB after operation

In this example, we open SQLite db for each command, so we should close db after each operation.

```
node index.js init
node index.js create "<title>" ["<date>"]
node index.js list
node index.js get <id>
node index.js update <id> "<title>" "<date>"
node index.js delete <id>
```

ensureTable

Before executing the SQL command `cb`, we create a posts table of (_id, title, date) if the table does not exist.

```
  PRAGMA foreign_keys = ON;
  CREATE TABLE IF NOT EXISTS posts (
    _id   INTEGER PRIMARY KEY,
    title TEXT NOT NULL,
    date  TEXT
  );`;
```

```
function ensureTable(cb) {
  const sql = `
  PRAGMA foreign_keys = ON;
  CREATE TABLE IF NOT EXISTS posts (
    _id   INTEGER PRIMARY KEY,
    title TEXT NOT NULL,
    date  TEXT
  );`;
  db.exec(sql, cb);
}
```

- db.exec(sql, cb) runs the SQL statements.
- When SQLite finishes executing, it calls your callback cb(err) automatically.
- This is a standard Node.js callback pattern.

initialization

It checks the creation of table.

- When there is an error it displays a error message and return.
- Otherwise, it closes the database.

```
function cmdInit() {
  ensureTable((err) => {
    if (err) return console.error('Init error:', err.message);
    console.log('Initialized DB and ensured posts table exists.');
    db.close();
  });
}
```

Create

1. Check if title is given.
2. Check if table exists, and run `INSERT INFO` is executed.

```
function cmdCreate(title, date) {
  if (!title) {
    console.error('Usage: node index.js create "<title>" ["<date>"]');
    return db.close();
  }
  ensureTable((err) => {
    if (err) {
      console.error('Create/init error:', err.message);
      return db.close();
    }
    const sql = 'INSERT INTO posts (title, date) VALUES (?, ?)';
    db.run(sql, [title, date || ''], function(err2) {
      if (err2) {
        console.error('Create error:', err2.message);
        return db.close();
      }
      console.log(JSON.stringify({ _id: this.lastID, title, date: date || '' }, null, 2));
      db.close();
    });
  });
}
```

- The date can be empty, but in this case, it should be replaced by empty string: `date || ''`.
- We should use normal function, not arrow function, because we need to use `this`.

```
    db.run(sql, [title, date || ''], function(err2) {
      if (err2) {
        console.error('Create error:', err2.message);
        return db.close();
      }
      console.log(JSON.stringify({ _id: this.lastID, title, date: date || '' }, null, 2));
      db.close();
    });
  });
```

Read All

- We use db.all to read all content.
- The returned rows becomes JSON string with `JSON.stringify`.

```
function cmdList() {
  ensureTable((err) => {
    if (err) { console.error('List/init error:', err.message);
      return db.close();
    }
    db.all('SELECT _id, title, date FROM posts ORDER BY _id ASC', [], (err2, rows) => {
      if (err2) { console.error('List error:', err2.message);
        return db.close();
      }
      console.log(JSON.stringify(rows, null, 2));
      db.close();
    });
  });
}
```

- We can process each row one by one using the forEach method.

```
db.all('SELECT * FROM users', [], (err, rows) => {
  rows.forEach(r => console.log(r));
});
```

Read One

- From the input argument, idStr, we get the row and transform it into JSON string.

```
function cmdGet(idStr) {
  const id = parseInt(idStr, 10);
  if (Number.isNaN(id)) {
    console.error('Usage: node index.js get <id>');
    return db.close();
  }
  ensureTable((err) => {
    if (err) { console.error('Get/init error:', err.message); return db.close(); }
    db.get('SELECT _id, title, date FROM posts WHERE _id = ?', [id], (err2, row) => {
      if (err2) { console.error('Get error:', err2.message); return db.close(); }
      if (!row) console.log('null');
      else console.log(JSON.stringify(row, null, 2));
      db.close();
    });
  });
}
```

Update

- Update title and date based on id.
- Check how many rows are updated.
- Get the updated row and make JSON string.

```
function cmdUpdate(idStr, title, date) {
  const id = parseInt(idStr, 10);
  if (Number.isNaN(id) || title === undefined || date === undefined) {
    console.error('Usage: node index.js update <id> "<title>" "<date>"');
    return db.close();
  }
  ensureTable((err) => {
    if (err) { console.error('Update/init error:', err.message); return db.close(); }
    const sql = 'UPDATE posts SET title = ?, date = ? WHERE _id = ?';
    db.run(sql, [title, date, id], function(err2) {
      if (err2) { console.error('Update error:', err2.message); return db.close(); }
      if (this.changes === 0) {
        console.log('null');
        return db.close();
      }
      db.get('SELECT _id, title, date FROM posts WHERE _id = ?', [id], (err3, row) => {
        if (err3) { console.error('Fetch after update error:', err3.message); return db.close(); }
        console.log(JSON.stringify(row, null, 2));
        db.close();
      });
    });
  });
}
```

Delete

- Delete the row with id = `id`.

```
function cmdDelete(idStr) {
  const id = parseInt(idStr, 10);
  if (Number.isNaN(id)) {
    console.error('Usage: node index.js delete <id>');
    return db.close();
  }
  ensureTable((err) => {
    if (err) { console.error('Delete/init error:', err.message); return db.close(); }
    db.run('DELETE FROM posts WHERE _id = ?', [id], function(err2) {
      if (err2) { console.error('Delete error:', err2.message); return db.close(); }
      console.log(JSON.stringify({ ok: true, deletedCount: this.changes }, null, 2));
      db.close();
    });
  });
}
```