Step 4: SQLite Database

From MongoDB to SQLite

What Changed?

- Step 1-3: Used **MongoDB** (NoSQL document database)
- Step 4: Uses **SQLite** (SQL relational database)
- Same application features, different database technology
- **Learning objective:** Understanding database portability

Why SQLite?

Advantages

- **No server needed** - Just a file
- **Zero configuration** - Works out of the box
- **Portable** - Single file database
- **Perfect for learning** - Simple setup
- **Good for small projects** - Fast and lightweight

When to Use

- Prototyping and development
- Small applications
- Mobile apps
- Embedded systems

MongoDB vs SQLite

Feature	MongoDB	SQLite
Type	NoSQL (Document)	SQL (Relational)
Data Format	JSON/BSON	Tables/Rows
Server	Required	File-based
Queries	JSON syntax	SQL syntax
Schema	Flexible	Fixed schema
Auto-increment	Manual	Built-in

Database Connection - Before

MongoDB:

```
import { MongoClient } from 'mongodb';

const client = new MongoClient(uri);
await client.connect();
const db = client.db('todoapp');
const posts = db.collection('posts');
```

- Needs connection URI
- Connects to remote/local server
- Uses collections

Database Connection - After

SQLite:

```
import sqlite3 from 'sqlite3';
import path from 'path';

const DB_PATH = path.join(__dirname, '../todoapp.sqlite');

export const db = new Database(DB_PATH, (err) => {
  if (err) {
    console.error('Failed to open database:', err.message);
    process.exit(1);
  }
  console.log('Connected to SQLite database');
});
```

- Uses local file path
- No server needed
- Creates file if not exists

Schema Definition - Before

MongoDB:

```
// No schema required
// Just insert data
const newPost = {
  _id: newId,
  title: req.body.title,
  date: req.body.date
};
await posts.insertOne(newPost);
```

- Schema-less (flexible)
- Define structure per document

Schema Definition - After

SQLite:

```
CREATE TABLE IF NOT EXISTS posts (
  _id   INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL,
  date  TEXT
);
```

- **Fixed schema** - Must define first
- **Data types** - INTEGER, TEXT, etc.
- **Constraints** - NOT NULL, PRIMARY KEY
- **AUTOINCREMENT** - Built-in ID generation

CRUD: Create - Before

MongoDB:

```
// Manual ID generation
const last = await posts.find({ _id: { $type: 'int' } })
  .sort({ _id: -1 }).limit(1).toArray();
const newId = last.length ? (last[0]._id + 1) : 1;

const doc = { _id: newId, title, date };
await posts.insertOne(doc);
```

- Need to calculate next ID manually
- JSON-based insert

CRUD: Create - After

SQLite:

```
const sql = 'INSERT INTO posts (title, date) VALUES (?, ?)';

db.run(sql, [title, date || ''], function (err) {
  if (err) {
    return res.status(500).json({ error: 'Failed to create post' });
  }

  // Auto-generated ID available
  const doc = { _id: this.lastID, title, date: date || '' };
  res.status(201).json(doc);
});
```

- **SQL query** with parameterized values (`?`)
- **AUTOINCREMENT** handles ID automatically
- `this.lastID` gives the new ID

CRUD: Read All - Before

MongoDB:

```
const posts = await db.collection('posts')
  .find({})
  .sort({ _id: 1 })
  .toArray();

res.json(posts);
```

- JSON query syntax
- Method chaining
- Promise-based

CRUD: Read All - After

SQLite:

```
const sql = 'SELECT _id, title, date FROM posts ORDER BY _id ASC';

db.all(sql, [], (err, posts) => {
  if (err) {
    return res.status(500).json({ error: 'Failed to fetch posts' });
  }
  res.json(posts);
});
```

- **SQL SELECT** statement
- **db.all()** for multiple rows
- Callback-based

CRUD: Read One - Before

MongoDB:

```
const doc = await db.collection('posts')
  .findOne({ _id: id });

if (!doc) {
  return res.status(404).json({ error: 'Not found' });
}
res.json(doc);
```

- `findOne()` method
- JSON query object

CRUD: Read One - After

SQLite:

```
const sql = 'SELECT _id, title, date FROM posts WHERE _id = ?';

db.get(sql, [id], (err, doc) => {
  if (err) {
    return res.status(500).json({ error: 'Failed to fetch post' });
  }
  if (!doc) {
    return res.status(404).json({ error: 'Not found' });
  }
  res.json(doc);
});
```

- **db.get()** for single row
- **WHERE** clause with parameter
- `?` prevents SQL injection

CRUD: Update - Before

MongoDB:

```
const result = await db.collection('posts').findOneAndUpdate(
  { _id: id },
  { $set: { title, date } },
  { returnDocument: 'after' }
);

if (!result.value) {
  return res.status(404).json({ error: 'Not found' });
}
res.json(result.value);
```

- `$set` operator
- Returns updated document

CRUD: Update - After

SQLite:

```
const sql = 'UPDATE posts SET title = ?, date = ? WHERE _id = ?';

db.run(sql, [title, date, id], function (err) {
  if (err) {
    return res.status(500).json({ error: 'Failed to update post' });
  }

  if (this.changes === 0) {
    return res.status(404).json({ error: 'Not found' });
  }

  // Fetch updated document
  db.get('SELECT _id, title, date FROM posts WHERE _id = ?', [id],
    (err2, doc) => res.json(doc));
});
```

CRUD: Delete - Before

MongoDB:

```
const result = await db.collection('posts')
  .deleteOne({ _id: id });

if (result.deletedCount === 0) {
  return res.status(404).json({ error: 'Not found' });
}

res.json({ ok: true, deletedId: id });
```

- `deleteOne()` method
- Check `deletedCount`

CRUD: Delete - After

SQLite:

```
const sql = 'DELETE FROM posts WHERE _id = ?';

db.run(sql, [id], function (err) {
  if (err) {
    return res.status(500).json({ error: 'Failed to delete post' });
  }

  if (this.changes === 0) {
    return res.status(404).json({ error: 'Not found' });
  }

  res.json({ ok: true, deletedId: id });
});
```

- **DELETE** SQL statement
- Check `this.changes` for affected rows

Database Methods Comparison

MongoDB

- `insertOne()` - Create
- `find().toArray()` - Read all
- `findOne()` - Read one
- `updateOne()` / `findOneAndUpdate()` - Update
- `deleteOne()` - Delete

SQLite

- `db.run()` - INSERT, UPDATE, DELETE
- `db.all()` - SELECT multiple rows
- `db.get()` - SELECT single row

Parameterized Queries

Always use `?` placeholders to prevent SQL injection:

```
// ✅ SAFE - Parameterized
const sql = 'SELECT * FROM posts WHERE _id = ?';
db.get(sql, [id], callback);

// ❌ DANGEROUS - String concatenation
const sql = `SELECT * FROM posts WHERE _id = ${id}`;
db.get(sql, callback);
```

Why? Prevents malicious SQL injection attacks

Async Patterns: Callbacks vs Promises

Why this matters:

- Different libraries use different async patterns
- You need to adapt your code accordingly
- Understanding both helps you work with any library

MongoDB: Native Promise Support

MongoDB driver **natively supports Promises** (modern approach):

```
// Works with async/await
const posts = await db.collection('posts').find({}).toArray();

// Also supports .then()
db.collection('posts').find({}).toArray()
  .then(posts => console.log(posts))
  .catch(err => console.error(err));
```

Why Promises?

- Cleaner code with `async/await`
- Better error handling
- Modern JavaScript standard

SQLite: Callback-based API

SQLite3 library uses **callbacks** (older pattern):

```
// Must use callbacks
db.all('SELECT * FROM posts', [], (err, posts) => {
  if (err) {
    return console.error(err);
  }
  console.log(posts);
});

// Cannot use await directly
// await db.all('SELECT * FROM posts', []);  // ❌ Won't work
```

Why Callbacks?

- SQLite3 library was created before Promises
- Library design choice

Making SQLite Work with Promises

Option 1: Use `util.promisify`

```
import { promisify } from 'util';

const dbAll = promisify(db.all.bind(db));
const posts = await dbAll('SELECT * FROM posts', []);
```

Option 2: Use `better-sqlite3` (synchronous, simpler)

```
import Database from 'better-sqlite3';
const db = new Database('todoapp.sqlite');

const posts = db.prepare('SELECT * FROM posts').all();
```

Why We Use Callbacks in Step 4

Educational reasons:

1. Learn **both async patterns** - callbacks and promises
2. Understand **library limitations** - not all libs support promises
3. Practice **adapting code** to different APIs
4. Appreciate **modern improvements** (async/await)

Real-world: You'll encounter both patterns in existing projects

Auto-increment Comparison

MongoDB - Manual

```
const last = await posts.find({ _id: { $type: 'int' } })
  .sort({ _id: -1 }).limit(1).toArray();
const newId = last.length ? (last[0]._id + 1) : 1;
```

SQLite - Built-in

```
CREATE TABLE posts (
  _id INTEGER PRIMARY KEY AUTOINCREMENT,
  ...
);
```

Access with `this.lastID` after INSERT

Running Application

Same as before:

```
npm install
npm start
```

New: Database file created automatically:

- File: `todoapp.sqlite`
- Location: Project root directory
- Can be opened with SQLite browser tools

Key Concepts

- **SQL vs NoSQL** - Different query paradigms
- **Schema Definition** - Fixed vs flexible structure
- **Parameterized Queries** - Prevent SQL injection
- **File-based Database** - No server required
- **Built-in Auto-increment** - Simpler ID generation
- **Callbacks vs Promises** - Different async patterns

When to Use Which?

Use MongoDB when:

- Schema changes frequently
- Need horizontal scaling
- Working with complex nested data
- Large distributed systems

Use SQLite when:

- Simple CRUD applications
- Prototyping/development
- Small to medium data sets
- No concurrent writes needed