**Guía Extensiva: Integración de Node.js con SQLite3**

Esta guía práctica y extendida te enseñará a conectar y manipular una base de datos SQLite directamente desde tus aplicaciones de Node.js. Este es un paso fundamental para crear aplicaciones que necesiten persistir datos, como sistemas de usuarios, blogs, inventarios y mucho más, sin la complejidad de un gran servidor de base de datos.

**Requisitos Previos:**

* Tener Node.js y npm instalados.
* Conocimientos básicos de JavaScript (variables, funciones, objetos, arrays, callbacks).

**Configuración Inicial del Proyecto:**

1. Crea una nueva carpeta para tu proyecto (ej: mi-app-sqlite).
2. Abre una terminal en esa carpeta.
3. Inicia un proyecto de Node.js: npm init -y
4. Instala el paquete sqlite3: npm install sqlite3

¡Ya estás listo para comenzar!

**1. Conexión y Creación de la Base de Datos**

**Documentación y Explicación**

**SQLite** es una base de datos "basada en archivos", lo que significa que toda la base de datos se almacena en un único archivo en tu computadora. El primer paso siempre es establecer una conexión con este archivo.

* **require('sqlite3')**: Importa el módulo sqlite3 a nuestro script para poder utilizar sus funcionalidades. Usar .verbose() (require('sqlite3').verbose()) es una buena práctica durante el desarrollo, ya que proporciona mensajes de error más detallados y trazas de la pila (stack traces), facilitando la depuración.
* **new sqlite3.Database(ruta, [modo], callback)**: Este es el constructor que crea un objeto de base de datos y abre la conexión.
* ruta: Un string con la ruta al archivo de la base de datos (ej: ./empresa.db). Si el archivo no existe, SQLite lo creará automáticamente.
* [modo] (opcional): Permite especificar permisos, como abrir en modo de solo lectura.
* callback: Una función que se ejecuta una vez que la conexión se ha establecido (o ha fallado). Recibe un objeto err como primer argumento. Si err es null, la conexión fue exitosa.

**Ejemplo de Conexión:**

const sqlite3 = require('sqlite3').verbose();  
  
// Conectarse a la base de datos. El archivo 'mi\_base\_de\_datos.db' se creará si no existe.  
const db = new sqlite3.Database('./mi\_base\_de\_datos.db', (err) => {  
  if (err) {  
    // No se pudo abrir la conexión  
    console.error("Error al conectar con la base de datos:", err.message);  
  } else {  
    // Conexión exitosa  
    console.log('Conectado a la base de datos SQLite.');  
  }  
});

* **db.run(sql, [params], callback)**: Ejecuta una sentencia SQL que no devuelve filas de datos (como CREATE, INSERT, UPDATE, DELETE).
* sql: El string con la consulta SQL.
* [params] (opcional): Un array de parámetros para las consultas parametrizadas.
* callback: Una función que se ejecuta al finalizar. Recibe un err si algo salió mal.

**Ejercicios de Conexión y Creación**

Ejercicio 1.1: Crear la Tabla 'usuarios'

Tarea: En un nuevo archivo app.js, escribe el código para:

1. Conectarte a una base de datos llamada empresa.db.
2. Crear una tabla llamada usuarios si no existe (CREATE TABLE IF NOT EXISTS).
3. La tabla debe tener las siguientes columnas:

* id (INTEGER, PRIMARY KEY, AUTOINCREMENT)
* nombre (TEXT, NOT NULL)
* email (TEXT, NOT NULL, UNIQUE)

Ejercicio 1.2: Crear una Tabla 'productos'

Tarea: En el mismo script, añade el código para crear una segunda tabla llamada productos.

1. La tabla debe tener las siguientes columnas:

* id (INTEGER, PRIMARY KEY, AUTOINCREMENT)
* nombre (TEXT, NOT NULL)
* precio (REAL, NOT NULL)
* stock (INTEGER, DEFAULT 0)

**2. Insertando Datos (Operaciones CREATE)**

**Documentación y Explicación**

Para insertar datos, también usamos db.run(). Es crucial utilizar **consultas parametrizadas** para prevenir ataques de inyección de SQL. Se usan ? como marcadores de posición en la consulta SQL, y luego se pasa un array con los valores correspondientes.

* **function(err)**: El callback de db.run es una función regular (no una función de flecha) si necesitas acceder a this.
* this.lastID: Contiene el id de la última fila insertada (muy útil para saber el ID autoincremental que se generó).
* this.changes: Contiene el número de filas que fueron afectadas por la consulta.

**Ejemplo de Inserción:**

const sql = 'INSERT INTO usuarios (nombre, email) VALUES (?, ?)';  
const params = ['Juan Pérez', 'juan.perez@example.com'];  
  
db.run(sql, params, function(err) {  
  if (err) {  
    return console.error("Error al insertar:", err.message);  
  }  
  console.log(`Fila insertada con ID: ${this.lastID}`);  
});

**Ejercicios de Inserción**

Ejercicio 2.1: Insertar un Usuario

Tarea: Añade el código para insertar un único usuario en la tabla usuarios: nombre "Ana López", email "ana.lopez@example.com".

Ejercicio 2.2: Insertar Múltiples Usuarios

Tarea: Crea un array de objetos, donde cada objeto representa un usuario. Luego, usa un bucle (como forEach) para recorrer el array e insertar cada usuario en la base de datos.

Array de Ejemplo:

const nuevosUsuarios = [  
  { nombre: "Carlos García", email: "carlos.garcia@example.com" },  
  { nombre: "Laura Martínez", email: "laura.martinez@example.com" },  
  { nombre: "Pedro Rodríguez", email: "pedro.rodriguez@example.com" }  
];

Ejercicio 2.3: Insertar Productos

Tarea: De manera similar al ejercicio anterior, crea un array de objetos productos e insértalos en la tabla productos que creaste en el ejercicio 1.2.

Array de Ejemplo:

const nuevosProductos = [  
  { nombre: "Laptop Gamer", precio: 1250.75, stock: 15 },  
  { nombre: "Teclado Mecánico", precio: 150.50, stock: 30 },  
  { nombre: "Monitor 27 pulgadas", precio: 300.00, stock: 25 }  
];

**3. Consultando Datos (Operaciones READ)**

**Documentación y Explicación**

Para leer datos, sqlite3 ofrece varios métodos. Elegir el correcto mejora la eficiencia de tu aplicación.

* **db.all(sql, params, callback)**: Ejecuta la consulta y pasa un array con **todas las filas** del resultado al callback (err, rows). Es ideal para resultados pequeños o medianos donde necesitas todos los datos a la vez.  
  **Ejemplo db.all:**  
  db.all("SELECT id, nombre FROM usuarios", [], (err, rows) => {  
    if (err) throw err;  
    rows.forEach(row => console.log(row.nombre));  
  });
* **db.get(sql, params, callback)**: Ejecuta la consulta y devuelve solo la **primera fila** que encuentra al callback (err, row). Si no encuentra ninguna, row es undefined. Es más eficiente que db.all para buscar un único resultado.  
  **Ejemplo db.get:**  
  db.get("SELECT \* FROM usuarios WHERE id = ?", [1], (err, row) => {  
    if (err) throw err;  
    console.log(row ? row.nombre : "Usuario no encontrado");  
  });
* **db.each(sql, params, rowCallback, completeCallback)**: Es la forma más eficiente para grandes conjuntos de datos. Llama a rowCallback **una vez por cada fila**, sin cargar todo en memoria. Al final, llama a completeCallback.  
  **Ejemplo db.each:**  
  db.each("SELECT nombre FROM usuarios", [], (err, row) => {  
    if (err) throw err;  
    console.log(`Procesando usuario: ${row.nombre}`);  
  }, (err, count) => {  
    if (err) throw err;  
    console.log(`Se procesaron ${count} filas.`);  
  });

**Ejercicios de Consulta**

Ejercicio 3.1: Obtener Todos los Usuarios

Tarea: Escribe el código para consultar y mostrar en la consola todos los usuarios de la tabla usuarios, ordenados por nombre. Usa db.all().

Ejercicio 3.2: Obtener un Usuario por su ID

Tarea: Escribe el código para buscar un usuario específico por su id. Usa db.get() para esta tarea. Prueba buscando un ID que exista y otro que no.

Ejercicio 3.3: Buscar Usuarios por Dominio de Email

Tarea: Encuentra a todos los usuarios cuyo email termine en @example.com.

Pista: Usa el operador LIKE de SQL con el comodín %.

Ejercicio 3.4: Buscar Productos por Rango de Precio

Tarea: Encuentra todos los productos cuyo precio esté entre 100 y 500.

Pista: Usa la cláusula WHERE precio BETWEEN ? AND ?.

Ejercicio 3.5: Procesar Usuarios uno por uno

Tarea: Imagina que tienes miles de usuarios y quieres imprimir un saludo para cada uno. Para no cargar todos los datos en memoria, usa db.each(). Para cada usuario, imprime "Hola, [nombre del usuario]". Cuando termines, imprime "Se ha saludado a todos los usuarios."

**4. Actualizando Datos (Operaciones UPDATE)**

**Documentación y Explicación**

Para actualizar registros, usamos db.run(). Es crucial incluir una cláusula WHERE para especificar qué filas modificar. El callback nos informa sobre el resultado de la operación a través de this.changes.

**Ejemplo de Actualización:**

const sql = "UPDATE productos SET precio = ? WHERE id = ?";  
db.run(sql, [1300.00, 1], function(err) {  
  if (err) return console.error(err.message);  
  console.log(`Filas actualizadas: ${this.changes}`);  
});

**Ejercicios de Actualización**

Ejercicio 4.1: Actualizar el Email de un Usuario

Tarea: Elige un usuario de tu base de datos (por ejemplo, el que tiene id = 2) y actualiza su email a un nuevo valor. Imprime un mensaje en la consola indicando si la actualización fue exitosa.

Ejercicio 4.2: Incrementar Stock de un Producto

Tarea: Elige un producto y aumenta su stock en 10 unidades.

Pista: La consulta sería UPDATE productos SET stock = stock + ? WHERE id = ?.

**5. Eliminando Datos (Operaciones DELETE)**

**Documentación y Explicación**

Para eliminar datos, db.run() es de nuevo nuestro método. La cláusula WHERE es **extremadamente importante** para evitar borrar datos no deseados.

**Ejemplo de Eliminación:**

const sql = "DELETE FROM usuarios WHERE id = ?";  
db.run(sql, [3], function(err) {  
  if (err) return console.error(err.message);  
  console.log(`Filas eliminadas: ${this.changes}`);  
});

**Ejercicios de Eliminación**

Ejercicio 5.1: Eliminar un Usuario por Email

Tarea: Elige un usuario (por ejemplo, el que tiene el email "pedro.rodriguez@example.com") y elimínalo de la base de datos. Verifica que se haya eliminado volviendo a consultar todos los usuarios.

Ejercicio 5.2: Eliminar Productos sin Stock

Tarea: Elimina todos los productos de la tabla productos cuyo stock sea igual a 0.

**6. Cerrando la Conexión**

**Documentación y Explicación**

Una vez que has terminado todas tus operaciones, es una buena práctica cerrar la conexión para liberar el archivo de la base de datos y otros recursos. En un script que ejecuta varias operaciones asíncronas, el cierre debe hacerse en el callback de la última operación.

db.close((err) => {  
  if (err) return console.error(err.message);  
  console.log('Conexión a la base de datos cerrada.');  
});

Ejercicio 6.1: Integrar el Cierre de Conexión

Tarea: Revisa tu archivo app.js. Asegúrate de que, después de que se completen todas las operaciones asíncronas, se llame a db.close().

**7. Ejercicios Finales de Integración**

Ejercicio Final 7.1: Sistema de Blog Básico

Tarea:

1. Crea una nueva tabla llamada posts con las siguientes columnas:

* id (INTEGER, PRIMARY KEY, AUTOINCREMENT)
* titulo (TEXT, NOT NULL)
* contenido (TEXT, NOT NULL)
* autor\_id (INTEGER), con una FOREIGN KEY que referencia al id de la tabla usuarios.

1. Inserta un par de posts, asegurándote de que el autor\_id corresponda a IDs de usuarios existentes.
2. Escribe una consulta que obtenga todos los posts y, para cada post, muestre también el nombre del autor en lugar de solo su ID.  
   Pista: Necesitarás usar una consulta JOIN. SELECT posts.titulo, posts.contenido, usuarios.nombre AS nombre\_autor FROM posts JOIN usuarios ON posts.autor\_id = usuarios.id;

Ejercicio Final 7.2: Refactorización a Promesas

Tarea: El estilo de sqlite3 con callbacks puede llevar al "Callback Hell". Un enfoque más moderno es "envolver" estas funciones en Promesas.

1. Crea una nueva función query(sql, params) que devuelva una Promise.
2. Dentro de la promesa, usa db.all(sql, params, ...)

* Si hay un error, reject(err).
* Si tiene éxito, resolve(rows).

1. Reescribe el ejercicio 3.1 (Obtener Todos los Usuarios) usando tu nueva función query con .then() o async/await.

**Pista para la función query:**

// function query(sql, params = []) {  
//   return new Promise((resolve, reject) => {  
//     db.all(sql, params, (err, rows) => {  
//       // Tu lógica de resolve/reject aquí  
//     });  
//   });  
// }

Ejercicio Final 7.3: Búsqueda Avanzada con JOIN

Tarea: Escribe una consulta que encuentre todos los posts escritos por un autor específico, buscándolo por su email.

Pista: Necesitarás un JOIN y una cláusula WHERE que filtre por usuarios.email.