Node.js & MySQL Mastery Guide

A Beginner's Reference for Async/Await and Database Pools

1 The Concepts: Handling Time (Asynchronous Code)

JavaScript is single-threaded. It can only do one thing at a time. When we talk to a database, it takes time (milliseconds or seconds). We need a way to "wait" without stopping the whole server.

1.1 A. The Old Way: Callbacks (Avoid This)

This is like ordering a pizza and standing at the counter until it is done. You cannot do anything else.

- **Visual:** `function(error, result)`
- **Problem:** If you need to do 3 things in a row, you get "Callback Hell" (nested code that is hard to read).
- **Status: Deprecated** for modern flow control. Do not mix this with `await`.

1.2 B. The Modern Way: Async / Await

This is like ordering a pizza, getting a buzzer, and sitting down. You can check your phone (do other work). When the buzzer rings (`await` finishes), you go get the pizza.

- **`async`**: Put this before a function definition to tell Node.js "This function contains waiting steps."
- **`await`**: Put this before a database call. It pauses *that specific function* until the data arrives.

2 The Setup: High-Performance Database Config

For real-world apps (like your Driver App), you never create a single connection. You create a **Pool**.

Analogy:

- **Single Connection:** A store with only 1 checkout lane. If 5 people come, 4 have to wait outside.
- **Pool:** A store with 100 checkout lanes. 100 people can check out at the exact same time.

2.1 Recommended `config/database.js`

Save this file for every project.

```
import mysql from 'mysql2/promise'; // CRITICAL: Use /promise
import dotenv from 'dotenv';
dotenv.config();

const pool = mysql.createPool({
    host: process.env.DB_HOST,
    user: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_NAME,

    // --- REAL WORLD SCALING SETTINGS ---
    waitForConnections: true,  // If all lanes full, queue the user
    connectionLimit: 50,       // Max simultaneous connections (lanes)
    queueLimit: 0,             // Unlimited queue size

    // --- KEEP ALIVE (Prevents "Protocol Enqueue After Fatal Error") ---
    enableKeepAlive: true,
    keepAliveInitialDelay: 0
});

export default pool;
```

3 Cheat Sheet: How to Write Queries

In `mysql2/promise`, every query returns an **Array**: `[rows, fields]`.

- `rows`: The actual data (results).
- `fields`: Technical info about columns (usually ignored).

We use **Destructuring** to get just the rows: `const [rows] = ...`

3.1 A. SELECT (Getting Data)

Scenario: Get a driver's profile.

```
// Correct
const [rows] = await db.execute('SELECT * FROM drivers WHERE id = ?', [driverId
   ]);

if (rows.length === 0) {
    console.log("Driver not found");
} else {
    const driver = rows[0]; // Get the first result
    console.log("Driver Name:", driver.name);
}
```

3.2 B. INSERT (Saving Data)

Scenario: Adding a refuel entry.

```javascript
const sql = 'INSERT INTO refuel (driver_id, amount) VALUES (?, ?)';
const [result] = await db.execute(sql, [101, 500]);

console.log("New ID created:", result.insertId); // The ID of the row just added
console.log("Rows affected:", result.affectedRows); // Should be 1
```

3.3 C. UPDATE (Changing Data)

Scenario: Approving a refuel request.

```javascript
const sql = 'UPDATE refuel SET status = ? WHERE id = ?';
const [result] = await db.execute(sql, ['Approved', 55]);

if (result.affectedRows === 0) {
    console.log("Error: No record found with ID 55");
} else {
    console.log("Success: Record updated");
}
```

4 Real World Pattern: Transactions

In real apps, sometimes you need to do **two things at once**, or **nothing at all**.
Example: A driver pays for fuel.

1. Deduct money from Wallet table.
2. Add record to Expense table.

If step 2 fails, you MUST cancel step 1 (give money back). This is called a **Transaction**.

```javascript
// Example of a Real-World Transaction Wrapper
router.post('/pay-fuel', async (req, res) => {
    const connection = await db.getConnection(); // Get a dedicated lane

    try {
        await connection.beginTransaction(); // Start tracking changes

        // Step 1: Deduct Wallet
        const [updateRes] = await connection.execute(
            'UPDATE wallet SET balance = balance - ? WHERE driver_id = ?',
            [500, 101]
        );

        if (updateRes.affectedRows === 0) {
            throw new Error("Driver not found or insufficient funds");
        }

        // Step 2: Log Expense
        await connection.execute(
            'INSERT INTO expenses (driver_id, amount) VALUES (?, ?)',
            [101, 500]
        );

        await connection.commit(); // SAVE EVERYTHING PERMANENTLY
        res.json({ success: true });

    } catch (error) {
        await connection.rollback(); // ERROR? UNDO EVERYTHING
        console.error("Transaction failed:", error);
        res.status(500).json({ message: "Payment failed" });
    } finally {
        connection.release(); // Free up the lane for the next person
    }
});
```

5 Common Beginner Mistakes to Avoid

1. **Missing `await`**:
   Wrong: `const rows = db.execute(...)` → rows will be a Promise object, not data.
   Right: `const [rows] = await db.execute(...)`
2. **Using Callbacks with Promises**:
   Wrong: `await db.execute(sql, params, (err, res) => { ... })`
   Why: The library ignores the function at the end. The code inside `{ ... }` never runs.
3. **SQL Injection (Security Risk)**:
   Wrong: `db.execute('SELECT * FROM users WHERE name = ' + req.body.name)`
   Right: `db.execute('SELECT * FROM users WHERE name = ?', [req.body.name])`
   Why: The ? ensures hackers cannot insert malicious code into your database.

6 Summary Checklist for New Files

- Import db from your config.
- Make your route function `async (req, res)`.
- Use `try { ... } catch (error) { ... }` blocks.
- Use `const [data] = await db.execute(sql, [values])`.
- Send responses using `res.json()`.