SQL Connectivity with Node.js

Direct SQL Queries for CRUD Operations

Introduction

What is SQL Connectivity?

- SQL Connectivity allows applications to interact with relational databases like MySQL, MariaDB, or PostgreSQL.
- In Node.js, this is achieved using direct SQL queries, bypassing ORM (Object-Relational Mapping).

Why Avoid ORM?

- Greater control over raw SQL queries.
- Improved understanding of database operations.
- Optimized performance for specific queries

Architecture Overview

Architecture Layout:

- app.js → Main server configuration
- db.js → Database connection setup
- routes/ → API route definitions
- controllers/ → Logic to handle API requests
- services/ → SQL query execution

Data Flow:

Client Request → Route → Controller → Service → Database → Response

Database Connection Setup

db.js: Setting up MySQL connection

```
const mysql = require('mysql2');
const db = mysql.createConnection({
  host: 'localhost',user: 'root',password: '123',database: 'car_db',
});
db.connect((err) => {
  if (err) {console.error('Database connection failed:', err.message);}
else {console.log('Connected to MySQL Database');}
});
module.exports = db;
```

CRUD Operations with SQL Queries

Create Operation:

- const sql = 'INSERT INTO cars SET ?';
- db.query(sql, newCar, callback);

Read Operation:

- const sql = 'SELECT * FROM cars';
- db.query(sql, callback);

Update Operation:

- const sql = 'UPDATE cars SET ? WHERE id = ?';
- db.query(sql, [updatedCar, id], callback);

Delete Operation:

- const sql = 'DELETE FROM cars WHERE id = ?';
- db.query(sql, [id], callback);

Summary

- Direct SQL connectivity gives more control and insight into database operations.
- Node.js with MySQL allows for efficient CRUD operations without ORM overhead.
- Clear architecture with separation of concerns: Routes, Controllers, Services.