Grading Rubric: Node.js Express & MySQL2 CRUD (MVC Architecture)

Assignment: Build a CRUD REST API/Application using Node.js, Express, and MySQL2 following the MVC pattern.

Criteria	**Distinguished**	**Proficient**	**Developing**	**Needs Improvement**
1. Configuration & Security (.env)	• **Secure:** .env file is used for **ALL** sensitive data (DB host, user, password, port). • **Git Safety:** .gitignore exists and explicitly excludes .env and node_modules. • **Setup:** Project runs immediately via npm install and npm start without config errors.	• **Mostly Secure:** Uses .env but commits it to version control (Git). • **Partial Config:** Some credentials are strictly in .env, but non-sensitive config is hardcoded. • **Setup:** Requires minor manual tweaking to run.	• **Insecure:** Database credentials (password/user) are hardcoded directly into the JavaScript files. • **Missing Files:** .gitignore is missing or empty. • **Setup:** Major setup issues prevents running.	• **Unsafe:** No .env usage; credentials exposed. • **Broken:** Application does not run. • **Dependencies:** package.json missing or incorrect.
2. MVC Architecture & Structure	• **Strict Separation:** Logic is perfectly isolated into models/ (DB queries), controllers/ (Logic), and routes/ (Endpoints). • **Entry Point:** server.js or app.js is clean, handling only middleware and route	• **Good Separation:** Folders exist (models, controllers, routes), but some logic leaks (e.g., SQL queries inside a controller instead of a model). • **Entry Point:** server.js is slightly cluttered with logic that belongs	• **Weak Separation:** "MVC" folders exist, but files are empty or misused (e.g., routes doing DB queries directly). • **Monolithic:** Significant logic remains in the main entry file.	• **No Architecture:** All code (routes, DB connection, logic) is dumped into a single file. • **Disorganized:** No clear folder structure used.

	imports. • **Modularity:** Controller functions are exported and imported correctly.	elsewhere.		
3. Database Interaction (MySQL2)	• **Efficient:** Uses mysql.createPool for connection handling. • **Security:** Uses **Prepared Statements** (? placeholders) for ALL inputs to prevent SQL Injection. • **Modern Syntax:** Uses async/await with promise() wrapper for clean, non-blocking DB calls.	• **Functional:** Uses createConnection (single connection) instead of a pool. • **Mostly Secure:** Most queries use placeholders, but 1-2 minor queries might use string interpolation. • **Callbacks:** Uses traditional callbacks instead of promises (functional but dated).	• **Inefficient:** Opens and closes a connection manually for every single request. • **Vulnerable:** Uses template literals (e.g., ${id}) inside SQL queries (SQL Injection risk). • **Blocking:** Synchronous code blocks the event loop.	• **Broken DB:** Database connection fails. • **Syntax Errors:** SQL syntax is incorrect. • **No SQL:** Uses a different method (like an ORM) when raw MySQL2 was required.
4. CRUD Functionality & Routes	• **Complete:** CREATE, READ (All & Single), UPDATE, and DELETE work perfectly. • **HTTP Verbs:** Correct usage (GET, POST, PUT/PATCH, DELETE).	• **Functional:** All CRUD operations work, but edge cases fail (e.g., updating a non-existent ID throws a 500 error). • **Verbs:** Minor misuse (e.g., using POST for delete).	• **Partial:** 1 or 2 operations (e.g., Update or Delete) are missing or non-functional. • **Logic Errors:** Updates affect all rows instead of one; Delete removes wrong item.	• **Incomplete:** 3+ operations missing. • **Non-Functional:** API endpoints do not trigger database changes. • **Crash:** Routes cause server crash.

	• **Response:** Returns correct JSON/View data (e.g., 201 Created, 200 OK, 204 No Content).	• **Response:** Returns success but lacks meaningful data (e.g., missing ID of created item).	• **Verbs:** Everything is a GET or POST request.	
5. Error Handling & Quality	• **Robust:** Uses try...catch blocks for all async operations. • **Status Codes:** Returns accurate HTTP codes (404 Not Found, 400 Bad Request, 500 Internal Error). • **Feedback:** Provides clear, descriptive error messages to the client (not raw SQL errors).	• **Basic:** Generic error handling (everything returns 500). • **Console:** Logs errors to console but client request hangs (timeout). • **Codes:** Returns 200 OK even when an error occurs.	• **Minimal:** App crashes on invalid input. • **Silent Failures:** Failed queries return empty bodies with no indication of error. • **Trace:** Exposes raw stack traces or SQL errors to the user.	• **None:** No error handling implemented. • **Dirty Code:** Code is unreadable, poorly indented, or full of commented-out blocks.

Mandatory Checklist (Pass/Fail)

If any of the following are unchecked, the assignment may receive a zero or require resubmission.

- [ ] **.env File:** Is the .env file present locally but **excluded** from Git?
- [ ] **SQL Injection:** Are Prepared Statements used? (No VAR = string concatenation allowed).
- [ ] **Dependencies:** Are mysql2 and dotenv installed and used?
- [ ] **Run Command:** Does npm start (or node server.js) launch the server successfully?