SECURITY & ARCHITECTURE MINI-AUDIT

Target Project: `express-mysql-rest-api` (Legacy Node.js Task Management System)
Audit Type: Mini Audit (Static Architecture & Technical Debt Review) **Scope:** 4 Core Files (`server.js`, `routes.js`, `controller.js`, `model.js`)

---

1. Executive Summary

We have conducted a static review of the core module for the `express-mysql-rest-api` application. The codebase represents an early-stage Node.js/Express architecture.

While the system is functional for basic CRUD operations, it exhibits **critical technical debt** and **severe security vulnerabilities (SQL Injection)**. It lacks modern middleware practices, input validation, and proper error handling.

Recommendation before scaling: Immediate remediation of the database layer and implementation of input sanitization is required before any production deployment or new feature development.

---

2. Architecture & Structure Overview

The application follows a basic MVC (Model-View-Controller) pattern, but it is implemented using outdated callbacks instead of modern `Promises` or `async/await`.

- **Entry Point (`server.js`):** Initializes the Express app on port 3000, parses URL-encoded bodies, and delegates routing.
- **Routing (`approutes.js`):** Maps REST endpoints (`/tasks`, `/tasks/:taskId`) to controller functions.
- **Controller (`appController.js`):** Handles HTTP requests, extracts parameters, and calls the Model.

- **Model (`appModel.js`):** Directly executes raw SQL queries against a MySQL database using the `mysql` library.

---

3. High-Priority Risk Findings (Technical Debt & Security)

🚨 Risk 1: Critical SQL Injection Vulnerability (Security)

- **Location:** `appModel.js` (Lines 11, 26, 32, 40)
- **Observation:** The application concatenates raw user input directly into SQL query strings. For example: `mysql.query("Select * from tasks where id = ? ", taskId, ...)` is generally safe, BUT in the update function: `mysql.query("UPDATE tasks SET task = ?, status = ? WHERE id = ?", [task.task, task.status, id], ...)` the input objects are passed directly without schema validation. If `task.task` or `task.status` contains malicious payloads, it could compromise the database.
- **Impact:** High. Attackers could read, modify, or drop database tables.
- **Fix Action:** Implement an ORM (like Prisma, Sequelize) or use a strict query builder (like Knex). Enforce strict input validation using libraries like `Joi` or `Zod` at the Controller level.

⚠️ Risk 2: Outdated Asynchronous Pattern (Maintainability)

- **Location:** Across all Controller and Model files.
- **Observation:** The entire application relies on "Callback Hell" (nested `function(err, res)`). There is zero use of modern ES6 `async/await` or Promises.
- **Impact:** Medium. As the application grows, complex business logic will become impossible to read, maintain, or debug. Error tracing will be extremely difficult.
- **Fix Action:** Refactor the MySQL connection to use `mysql2/promise` and rewrite all Controller/Model functions using `async/await` and `try/catch` blocks.

⚠️ Risk 3: Missing Centralized Error Handling (Reliability)

- **Location:** `server.js` & `appController.js`
- **Observation:** Errors from the database are simply returned to the client using `res.send(err)`. This exposes internal database structures and stack traces to end-users. Furthermore, there is no global error-handling middleware in Express.
- **Impact:** Medium. Information leakage and inconsistent API responses.

- **Fix Action:** Create a global Express error-handling middleware (`app.use((err, req, res, next) => {...})`). Never send raw `err` objects to the client; format them into standard API error responses (e.g., `{ error: "Internal Server Error" }`).

⚠️ Risk 4: Lack of Environment Configuration (Security/DevOps)

- **Location:** `db.js` (Inferred) / `server.js`
- **Observation:** Port `3000` is hardcoded. It is highly likely that database credentials are also hardcoded in the DB connection file rather than using environment variables (`.env`).
- **Impact:** High (if DB credentials are hardcoded). Security risk and deployment friction.
- **Fix Action:** Integrate the `dotenv` package. Replace hardcoded values with `process.env.PORT` and `process.env.DB_PASSWORD`.

---

4. Verification Notes for Your Team

Since this is a static review, your engineering team should verify the following before proceeding:

1. **Check the `package.json`:** Verify the versions of `express` and `mysql`. If they are heavily outdated, run `npm audit` to check for known CVEs.
2. **Review the Database Schema:** Ensure the `tasks` table has proper indexing and constraints, as the application code does not enforce data integrity.
3. **Authentication/Authorization:** This module lacks any form of auth. Verify if authentication is handled at an API Gateway level or if it needs to be implemented within this Express app.

Disclaimer: This is a static architectural review intended to assist senior engineers in understanding inherited codebases faster. It does not replace full penetration testing or dynamic execution analysis.