Backend Track: Week 3 Assessment – Data Persistence & Basic Database Integration

This assessment is part of a 5-week Backend learning journey. Each assessment builds progressively, guiding learners from fundamentals to advanced concepts in Node.js development.

Scenario/Problem Statement

Your API from Week 2 successfully serves skills data from a static JSON file. However, in a real-world application, data needs to be dynamic and persistent. When the server restarts, any changes made to the in-memory data would be lost. To overcome this, you need to integrate a database that can reliably store and manage your application's data. Imagine your project idea from Week 3 (e.g., a task manager, a product catalog, or a list of project ideas) needs to store its details persistently. Your task is to extend your existing Express.js API to connect to a database and perform CRUD operations on your project idea data. This simulates the essential requirement of any modern web application: the ability to manage and persist data reliably.

Key Concepts Covered

- **Database Concepts:** Understanding the basics of relational databases (tables, rows, columns, primary keys, foreign keys) or NoSQL databases (documents, collections).
- **Database Integration:** Connecting Node.js/Express.js to a database.
- **SQL (Structured Query Language) Basics (for relational DBs):** Fundamental SQL commands for CRUD operations (SELECT, INSERT, UPDATE, DELETE).
- **ORM/ODM (Optional):** Introduction to Object-Relational Mappers (ORMs) like Sequelize or Object-Document Mappers (ODMs) like Mongoose for interacting

with databases in an object-oriented way.

- **CRUD Operations:** Implementing Create, Read, Update, and Delete functionalities through API endpoints that interact with the database.
- **Asynchronous Database Operations:** Handling database queries, which are inherently asynchronous, using Promises or `async/await`.
- **Error Handling:** Robust error handling for database connection issues and query failures.

Task Instructions

Participants will enhance their Express.js API to interact with a database for persistent storage of project idea details.

Step 1: Choose and Set Up Database

Choose ONE of the following options for your database setup:

Option A: SQLite (Recommended for simplicity and no external setup) 1. **Install `sqlite3`:** `bash npm install sqlite3` 2. **Create Database File:** Create a file named `database.sqlite` in your project root. SQLite will create this file if it doesn't exist. 3. **Database Initialization Script:** Create a script (e.g., `init-db.js`) to create your `ideas` table: ``` ``javascript const sqlite3 = require("sqlite3").verbose(); const db = new sqlite3.Database("./database.sqlite", (err) => { if (err) { console.error(err.message); throw err; } else { console.log("Connected to the SQLite database."); db.run(```CREATE TABLE ideas (id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT NOT NULL, description TEXT, status TEXT)`, (err) => { if (err) { // Table already created console.log("Table already exists."); } else { console.log("Table created."); // Insert some initial data const insert = "INSERT INTO ideas (title, description, status) VALUES (?,?,?)"; db.run(insert, ["Eco-Friendly Water Bottle", "A smart water bottle that tracks hydration and suggests refill stations.", "Concept"]); db.run(insert, ["AI-Powered Study Buddy", "An application that uses AI to create personalized study plans and quizzes.", "In Progress"]); } }); } });

~~~
 module.exports = db;
```
Run this script once: `node init-db.js`.
~~~

Option B: File-based JSON Database (e.g., `lowdb` or custom `fs` based) 1. **Install `lowdb`:** `bash npm install lowdb` 2. **Setup `lowdb` in `server.js`:** ```javascript const low = require("lowdb"); const FileSync = require("lowdb/adapters/FileSync"); const adapter = new FileSync("db.json"); // This will be your database file const db = low(adapter);

```
 // Set defaults if db.json is empty
db.defaults({ ideas: [] }).write();
```
Your `db.json` will be automatically created and managed by `lowdb`.
```

Step 2: Refactor API to Use Database

Modify your `server.js` (or create a new `routes/ideas.js` module) to interact with the chosen database instead of static JSON files.

1. **Database Connection:** Ensure your database connection (from Step 1) is accessible in your API routes.

2. **Implement GET /api/ideas:**

   - Retrieve all project ideas from the database.
   - Handle cases where no ideas are found.
   - Return a JSON array of ideas.

   `javascript // Example for SQLite app.get("/api/ideas", (req, res) => { db.all("SELECT * FROM ideas", [], (err, rows) => { if (err) { res.status(400).json({"error": err.message}); return; } res.json({ "message": "success", "data": rows }); }); });`

3. **Implement GET /api/ideas/:id:**

   - Retrieve a single project idea by its ID from the database.
   - Handle cases where the idea is not found (return 404).

   `javascript // Example for SQLite app.get("/api/ideas/:id", (req, res) => { const id = req.params.id; db.get("SELECT * FROM ideas WHERE id = ?", id, (err, row) => { if (err) { res.status(400).json({"error": err.message}); return; } if (!row) { return`
```

```
res.status(404).json({"message": "Idea not found"}); } res.json({
"message": "success", "data": row }); }); });
```

4. **Implement POST /api/ideas:**
   - Receive new project idea data from the request body.
   - Insert the new idea into the database.
   - Return the newly created idea (with its generated ID).

   ```
   javascript // Example for SQLite app.post("/api/ideas", (req, res) =>
   { const { title, description, status } = req.body; if (!title) {
   return res.status(400).json({"error": "Title is required"}); } const
   insert = "INSERT INTO ideas (title, description, status) VALUES
   (?,?,?)"; db.run(insert, [title, description, status || "Concept"],
   function (err) { if (err) { res.status(400).json({"error":
   err.message}); return; } res.status(201).json({ "message": "success",
   "data": { id: this.lastID, title, description, status: status ||
   "Concept" } }); }); });
   ```

5. **Implement PUT /api/ideas/:id:**
   - Receive updated project idea data and the ID from the request.
   - Update the corresponding idea in the database.
   - Return the updated idea.

   ```
   javascript // Example for SQLite app.put("/api/ideas/:id", (req, res)
   => { const id = req.params.id; const { title, description, status } =
   req.body; if (!title) { return res.status(400).json({"error": "Title
   is required"}); } db.run( `UPDATE ideas SET title =
   COALESCE(?,title), description = COALESCE(?,description), status =
   COALESCE(?,status) WHERE id = ?`, [title, description, status, id],
   function (err) { if (err) { res.status(400).json({"error":
   err.message}); return; } if (this.changes === 0) { return
   res.status(404).json({"message": "Idea not found"}); } res.json({
   "message": "success", "data": { id, title, description, status } });
   } ); });
   ```

6. **Implement DELETE /api/ideas/:id:**

  - Receive the ID of the idea to delete.
  - Remove the idea from the database.
  - Return a success message.

```
javascript // Example for SQLite app.delete("/api/ideas/:id", (req, res) => { const id = req.params.id; db.run("DELETE FROM ideas WHERE id = ?", id, function (err) { if (err) { res.status(400).json({"error": err.message}); return; } if (this.changes === 0) { return res.status(404).json({"message": "Idea not found"}); } res.status(200).json({"message": "Idea deleted successfully"}); }); });
```

Step 3: Test API Endpoints

1. **Start Server:** Run your Node.js server (`npm run dev` or `node server.js`).
2. **Test with Postman/cURL:** Use a tool like Postman, Insomnia, or `curl` to test all your CRUD endpoints:
    - `GET http://localhost:3001/api/ideas`
    - `GET http://localhost:3001/api/ideas/1`
    - `POST http://localhost:3001/api/ideas` (with JSON body: `{"title": "New Idea", "description": "A description", "status": "Pending"}`)
    - `PUT http://localhost:3001/api/ideas/1` (with JSON body: `{"title": "Updated Idea Title"}`)
    - `DELETE http://localhost:3001/api/ideas/1`

Step 4: Documentation and Submission

1. **Update `README.md`:** Update your `README.md` file to include:
    - Project description and purpose.
    - Instructions on how to set up and run the database (e.g., SQLite initialization).
    - Detailed API endpoint documentation for all CRUD operations, including request methods, URLs, expected request bodies, and example responses.
    - Technologies used (Node.js, Express.js, chosen database).

    - Challenges faced and solutions.
2. **Code Comments:** Ensure your database interaction logic and API routes are well-commented.

Deliverables

Participants are required to submit a single ZIP file containing the entire Node.js API project. The submission should include:

- The complete `skills-api` (or `ideas-api`) project folder.
- A comprehensive `README.md` file.
- All source code files (`server.js`, `init-db.js` if using SQLite, etc.).
- The database file (`database.sqlite` or `db.json`).

Evaluation Criteria

Assessments will be evaluated based on the following criteria, with a total of 100 points:

- **Database Setup & Connection (20 points):**
    - Successful setup and connection to the chosen database (SQLite or LowDB).
    - Proper database initialization and table/collection creation.
- **Read (GET) Operations (25 points):**
    - Successful implementation of `GET /api/ideas` (all ideas).
    - Successful implementation of `GET /api/ideas/:id` (single idea).
    - Correct data retrieval from the database.
- **Create (POST) Operation (20 points):**
    - Successful implementation of `POST /api/ideas`.
    - Correct insertion of new data into the database.
    - Proper handling of request body and response.
- **Update (PUT) Operation (15 points):**

    - Successful implementation of `PUT /api/ideas/:id`.
    - Correct updating of existing data in the database.
- **Delete (DELETE) Operation (10 points):**
    - Successful implementation of `DELETE /api/ideas/:id`.
    - Correct removal of data from the database.
- **Code Quality & Error Handling (10 points):**
    - Clean, readable, and well-organized code.
    - Appropriate error handling for database operations.
    - Consistent coding style.

This assessment is a cornerstone for backend development, enabling participants to build dynamic applications that can store and manage data persistently. Mastering database integration and CRUD operations is fundamental for creating robust and scalable web services.

Looking Ahead

In Week 4, you will delve into advanced database operations, including more complex queries and relationships, and crucially, implement robust authentication and authorization mechanisms to secure your API. Week 5 will then focus on preparing your backend application for production, covering deployment strategies, testing, and scalability considerations.

Recommended Resources

- **SQLite Official Documentation:** https://www.sqlite.org/docs.html
- **LowDB GitHub:** https://github.com/typicode/lowdb
- **SQL Basics Tutorial:** https://www.w3schools.com/sql/
- **Node.js `async/await`:** https://nodejs.dev/learn/understanding-javascript-asynchronous-programming
- **Express.js Routing:** https://expressjs.com/en/guide/routing.html

---