Month 3, Week 2: The Unabridged Learning Guide

Introduction: The Permanent Memory

In our last lesson, we built the architectural blueprint for our application's data using SQL's Data Definition Language (DDL). We now have a well-structured, relational schema waiting in our PostgreSQL database. However, a blueprint is not a building. Our Express application still lives in a world of temporary, in-memory arrays. It has no connection to this permanent memory we have so carefully designed.

This week, we build the bridge. We will connect our application to its database.

First, we will take a deep dive into **Advanced SQL**. We will move beyond simple `SELECT` statements and master the powerful tools that relational databases offer for querying complex, interconnected data. We will explore **`JOIN`s** to combine data from multiple tables, **`GROUP BY`** and aggregate functions to create powerful summary reports, and **subqueries** to handle complex logical conditions.

Second, we will take on the critical task of **integrating our Express.js application with PostgreSQL**. We will learn how to use the `pg` database driver, understand the non-negotiable importance of **connection pooling** for performance, and tackle the #1 security threat for database-driven applications: **SQL Injection**. We will learn why parameterized queries are the only acceptable way to interact with a database.

Finally, we will refactor our entire in-memory CRUD API, replacing every array method with a robust, secure, and persistent SQL query. By the end of this lesson, your application will have a permanent memory, and you will have the skills to architect a true, data-driven backend service.

Table of Contents

Module 1: Advanced SQL - The Art of the Query

The Power of Relationships: A `JOIN` Deep Dive

- **`INNER JOIN`:** The most common join. It returns only the rows where there is a match in **both** tables specified in the `ON` condition. If an author has no books, that author will *not* appear in the result of an inner join between `authors` and `books`.
- **`LEFT JOIN` (or `LEFT OUTER JOIN`):** This returns **all** rows from the "left" table (the first table mentioned) and the matching rows from the "right" table. If a row in the left table has no match in the right table, the columns from the right table will be filled with `NULL`. This is extremely useful for finding things that *don't* have a corresponding entry in another table (e.g., "find all users who have never written a post").
- **`RIGHT JOIN` (or `RIGHT OUTER JOIN`):** The inverse of a `LEFT JOIN`. It returns **all** rows from the "right" table and the matching rows from the "left" table. `NULL` is used for non-matches from the left side. In practice, `RIGHT JOIN` is rarely used; you can almost always rewrite the query as a `LEFT JOIN` by swapping the table order, which is often more intuitive to read.
- **`FULL OUTER JOIN`:** This returns **all** rows from **both** tables. If there is a match, the rows are combined. If a row in either table has no match in the other, it is still included, and the columns from the other table are filled with `NULL`.

Aggregating Data: `GROUP BY` & Aggregate Functions The `GROUP BY` clause is a powerful tool for creating summary reports. It collapses multiple rows into a single summary row based on the values in the specified column(s). It is almost always used with an aggregate function that performs a calculation on the group.

- `COUNT()`: Counts the number of rows in the group. `COUNT(*)` counts all rows, while `COUNT(column)` counts non-null values in that column.
- `SUM()`: Calculates the total sum of a numeric column.
- `AVG()`: Calculates the average value of a numeric column.
- `MAX()` / `MIN()`: Finds the maximum or minimum value in the group.

Example: E-commerce Report "For each product category, find the number of products in that category and the average price of a product in that category."

```
SELECT
    category,
    COUNT(*) AS number_of_products,
    AVG(price) AS average_price
FROM
    products
GROUP BY
    category;
```

Filtering Groups with `HAVING` The `HAVING` clause is a filter, just like `WHERE`, but it operates on the results of an aggregation. * `WHERE` is processed **before** `GROUP BY` and filters individual rows. * `HAVING` is processed **after** `GROUP BY` and filters the resulting groups.

Example: "Show me only the product categories that have more than 10 products."

```
SELECT
    category,
```

```
    COUNT(*) AS number_of_products
FROM
    products
GROUP BY
    category
HAVING
    COUNT(*) > 10;
```

You cannot use `WHERE COUNT(*) > 10` because `WHERE` does not know about the results of aggregate functions.

Subqueries: Queries within Queries A subquery allows you to perform a query and use its result as part of another query. They can be used in `WHERE`, `FROM`, and `SELECT` clauses.

Example: "Find all employees who earn more than the average salary of all employees." 1. First, you need to find the average salary: `SELECT AVG(salary) FROM employees`. 2. Then, you can use this result to filter the employees table.

A subquery combines this into one statement:

```
SELECT
    employee_name,
    salary
FROM
    employees
WHERE
    salary > (SELECT AVG(salary) FROM employees);
```

Logical Order of Operations in a `SELECT` Statement When you write a complex query, the database does not execute it in the order you typed it. Understanding the logical execution order is crucial for debugging and writing correct queries. 1. `FROM` and `JOIN`s: Determines the initial set of data. 2. `WHERE`: Filters individual rows. 3. `GROUP BY`: Groups the filtered rows into summary groups. 4. `HAVING`: Filters the summary groups. 5. `SELECT`: Selects the final columns to be returned. 6. `ORDER BY`: Sorts the final result set. 7. `LIMIT` / `OFFSET`: Paginates the final result set.

Module 2: Integrating Express.js with PostgreSQL

The Database Driver (`pg`) & Connection Pooling A "driver" is a piece of software that acts as a translator, allowing your application (Node.js) to communicate with a specific type of database (PostgreSQL) using a standardized API. The `pg` library is the official and most widely used driver for PostgreSQL in the Node.js ecosystem.

A **Connection Pool** is one of the most important performance optimizations in a backend application. Opening and closing a database connection is a slow, resource-intensive process involving network handshakes and authentication. A connection pool pre-opens a set of connections and keeps them ready. When your application needs to run a query, it "borrows" a connection from the pool. When the query is done, it "releases"

the connection back to the pool, making it immediately available for another part of your application. This avoids the setup cost for every query and is essential for handling concurrent user requests.

The #1 Security Threat: SQL Injection SQL Injection is a code injection technique used to attack data-driven applications. It occurs when malicious SQL statements are inserted into an entry field for execution (e.g., to dump the database contents to the attacker). This happens when you build your query strings by concatenating them with user-provided input.

Vulnerable Example:

```
const userInput = "1; DROP TABLE users; --"; // Malicious input
const query = `SELECT * FROM products WHERE id = ${userInput}`;
// Final query becomes: SELECT * FROM products WHERE id = 1; DROP TABLE users; --
```

The database will execute the `SELECT` statement, then happily execute the `DROP TABLE users` command, and the `--` comments out the rest of the original line. Your data is gone.

The Only Solution: Parameterized Queries You must **never** build queries with string concatenation. The only safe way is to use parameterized queries. This technique separates the SQL command from the data.

1. The SQL command is sent to the database with placeholders (like `$1`, `$2` in `pg`).
2. The database parses this query, understands its structure, and prepares a plan for execution.
3. Separately, the values (the user input) are sent to the database.
4. The database engine then safely inserts the values into the pre-planned query, ensuring they are treated only as data, not as executable SQL code.

Even if an attacker sends `1; DROP TABLE users; --` as input, the database will simply look for a product whose ID is the literal string `'1; DROP TABLE users; --'`, which it will not find. The malicious part of the string is never executed.

Refactoring the CRUD API: Step-by-Step The process involves replacing every in-memory array method with an `async` call to our database pool.

1. **Setup the Pool:** Create a `db/index.js` file to configure and export a single `Pool` instance for your entire application.
2. **Wrap in `async/try...catch`:** Every controller function that interacts with the database must become an `async` function. The database call (`await db.query(...)`) must be wrapped in a `try...catch` block. Any errors caught should be passed to Express's error handler with `next(err)`.
3. **Refactor `getAll`:** `const { rows } = await db.query('SELECT * FROM users'); res.json(rows);`
4. **Refactor `getById`:** `const { rows } = await db.query('SELECT * FROM users WHERE id = $1', [req.params.id]);`
5. **Refactor `create`:** `const { rows } = await db.query('INSERT INTO ... VALUES ($1, $2) RETURNING *', [value1, value2]);`
6. **Refactor `update`:** `const { rows } = await db.query('UPDATE ... SET column1 = $1 WHERE id = $2 RETURNING *', [newValue, id]);`

7. **Refactor `delete`:** `await db.query('DELETE FROM users WHERE id = $1', [id]);`

Take-Home Assessment: Refactoring the `posts` API

Objective: To solidify your understanding of SQL and database integration by refactoring a complete Express.js resource from using an in-memory array to a persistent PostgreSQL database.

The Task: You will be given the complete, multi-file Express.js project we built last week, which has a working CRUD API for `users` and `posts` using in-memory arrays. Your task is to refactor the **`posts` resource** to use the PostgreSQL database.

1. Create the Schema: * Write and execute a `CREATE TABLE` statement for a `posts` table that matches the data structure from last week. It should include an `id`, `title`, `content`, and a `user_id` column that has a `FOREIGN KEY` constraint referencing the `users` table.

2. Configure the Database Connection: * Create a `db/index.js` file and configure a connection pool for your PostgreSQL database.

3. Refactor the `postController.js` file: * `require` your database pool. * Go through every single function (`getAllPosts`, `getPostById`, `createPost`, `updatePost`, `deletePost`). * Convert each function to be `async`. * Replace all the array-based logic (`posts.find`, `posts.findIndex`, `posts.push`, `posts.splice`) with the appropriate parameterized SQL query using `await db.query(...)`. * Wrap all database logic in `try...catch` blocks and pass any errors to `next()`. * Ensure you are handling `404 Not Found` cases correctly when a query returns no rows.

Submission: Submit your entire refactored project (excluding `node_modules`) in a Pull Request to your personal assignments repository. ‘