Guided Project Guidelines

🛒 Supermarket API

You will build a **backend API** for a **Supermarket Management System**. This project will help you practice backend architecture, authentication, database modeling, CRUD operations, validation, error handling, and deployment.

You may use **Sequelize** or **Drizzle ORM** for database operations.

📂 Scenario – The Supermarket

Your backend should handle three main tables:

1. **Customers Table**
   - `customer_id (PK)`
   - `first_name`
   - `last_name`
   - `email`
   - `phone`
2. **Inventory Table**
   - `product_id (PK)`
   - `name`
   - `price`
   - `stock_quantity`
3. **Sales Table**
   - `sale_id (PK)`
   - `customer_id (FK → Customers)`
   - `product_id (FK → Inventory)`

- `sale_date`
- `quantity`
- `total_price`

Relationships:

- A **customer** can make many **sales**.
- Each **sale** involves one **product** from **inventory**.

📂 Project Guidelines

- **Project Structure**

  Organize your files cleanly:

  ```
  supermarket-api/
  ├── app.js          # Express setup
  ├── server.js       # Starts the server
  ├── routes/         # Route definitions
  ├── controllers/    # Request handlers
  ├── models/         # Database models (Drizzle/Sequelize)
  ├── middlewares/    # Auth, validation, logger
  └── .env            # Environment variables
  ```

- **Authentication**

  Implement a **basic login route** (`/login`) that uses **Basic Auth** with a hardcoded username and password (`admin / admin123`).

  - If the credentials are correct, return a **JWT**.
  - All other routes must be protected using middleware that verifies the JWT from the `Authorization: Bearer <token>` header.

- **CRUD Operations**

  Build routes to manage the three tables:

  - **Customers** → create, read, update, delete.

- **Inventory** → add new products, update stock, list items.
- **Sales** → record a sale, calculate total price (`quantity * price`), and reduce stock.

- **Validation & Error Handling**

  Use **Zod** or another validation library to check request bodies (e.g., prevent negative stock or missing fields).

  Wrap database queries in **try/catch** and return clear error messages (e.g., when a sale quantity is greater than available stock).

📝 Notes

- Keep your code modular: **controllers** handle logic, **routes** define endpoints, **models** define tables.
- Always require the JWT before performing database operations.

🧭 Route Map (by Router)

1) `authRouter` (base: `/auth`)

*All endpoints here are **public**. Everything else in the API requires a valid `Authorization: Bearer <JWT>` header.*

- **POST** `/auth/login`
  - **Purpose:** Exchange Basic Auth credentials for a JWT.
  - **Headers:** `Authorization: Basic <base64(admin:admin123)>`
  - **Body:** *(none)*
  - **Response:** `{ token: string, expiresIn: number }`
  - **Notes:** Hardcode `admin/admin123`. On success, sign a JWT with your `JWT_SECRET`.

> Apply verifyToken globally to all other routers (/customers, /inventory, /sales).

---

2) `customersRouter` (base: `/customers`)

Protected by JWT.

Core CRUD

- **POST** `/customers`
  - **Purpose:** Create a customer.
  - **Body (example):**

    ```
    { "first_name":"Ana", "last_name":"Lopez", "email":"ana@ex.com", "phone":"+504..." }
    ```

  - **Response:** Created customer record.
- **GET** `/customers`
  - **Purpose:** List customers (with optional pagination/search).
  - **Query (optional):** `q` (search by name/email), `limit`, `offset`, `sort` (e.g., `last_name`), `order` (`asc|desc`)
  - **Response:** `{ data: Customer[], total: number, limit, offset }`
- **GET** `/customers/:customer_id`
  - **Purpose:** Fetch one customer by ID.
  - **Response:** Customer record (or 404).
- **PUT** `/customers/:customer_id`
  - **Purpose:** Full update.
  - **Body (same shape as create)**
  - **Response:** Updated customer record.
- **DELETE** `/customers/:customer_id`

- **Purpose:** Delete customer (soft delete recommended).
- **Response:** `{ deleted: true }`

Convenience / Related

- **GET** `/customers/:customer_id/sales`
  - **Purpose:** List all sales for a specific customer.
  - **Response:** Sales array (joined with inventory).

---

3) `inventoryRouter` (base: `/inventory`)

Protected by JWT.

Core CRUD

- **POST** `/inventory`
  - **Purpose:** Add a product to inventory.
  - **Body (example):**

    ```
    { "name":"Milk 1L", "price":35.00, "stock_quantity":100 }
    ```

  - **Response:** Created product.
- **GET** `/inventory`
  - **Purpose:** List products (with filters).
  - **Query (optional):**
    - `q` (search by name)
    - `in_stock` (`true|false`)
    - `min_price`, `max_price`
    - `limit`, `offset`, `sort` (`name|price|stock_quantity`), `order` (`asc|desc`)
  - **Response:** `{ data: Product[], total, limit, offset }`
- **GET** `/inventory/:product_id`

- **Purpose:** Fetch a product by ID.
- **Response:** Product record (or 404).

- **PUT** `/inventory/:product_id`
  - **Purpose:** Full update (name, price, stock).
  - **Response:** Updated product.
- **DELETE** `/inventory/:product_id`
  - **Purpose:** Remove product (soft delete recommended).
  - **Response:** `{ deleted: true }`

Stock Utilities

- **POST** `/inventory/:product_id/adjust-stock`
  - **Purpose:** Increment or decrement stock safely.
  - **Body:** `{ "change": -3 }` (negative = reduce, positive = add)
  - **Response:** Updated product with new `stock_quantity`.

---

4) `salesRouter` (base: `/sales`)

Protected by JWT.

Core Operations

- **POST** `/sales`
  - **Purpose:** Create a sale (atomic: compute total, reduce stock).
  - **Body (example):**

    ```
    { "customer_id": 5, "product_id": 12, "quantity": 3, "sale_date": "2025-08-18" }
    ```

  - **Server logic:**
    1. Load product price & current `stock_quantity`.

2. Ensure `quantity <= stock_quantity` → otherwise **400** with message like `"Insufficient stock"`.
3. Compute `total_price = quantity * price`.
4. Insert sale.
5. Decrement stock (in a transaction).

- **Response:** Created sale record (including `total_price`).

- **GET** `/sales`
  - **Purpose:** List sales with optional filters.
  - **Query (optional):**
    - `customer_id`, `product_id`
    - `from`, `to` (date range)
    - `limit`, `offset`, `sort` (`sale_date|total_price|quantity`), `order`
  - **Response:** `{ data: Sale[], total, limit, offset }`
- **GET** `/sales/:sale_id`
  - **Purpose:** Fetch one sale (optionally expanded with customer + product).
  - **Query (optional):** `expand=customer,product`
  - **Response:** Sale record (joined if requested).
- **PUT** `/sales/:sale_id`
  - **Purpose:** Replace a sale (rare in real life; use carefully).
  - **Body:** Same shape as create.
  - **Notes:** If `quantity` changes, **reconcile stock** difference transactionally.
- **PATCH** `/sales/:sale_id`
  - **Purpose:** Partial update (e.g., fix `sale_date` or small adjustments).
  - **Notes:** If changing `quantity`, re-run stock reconciliation & `total_price`.
- **DELETE** `/sales/:sale_id`
  - **Purpose:** Cancel a sale.

- **Notes:** Optionally **restore stock** (transaction). Return `{ deleted: true }`.

Reporting

- **GET** `/sales/summary`
  - **Purpose:** High-level metrics.
  - **Response (examples):**
    - By product: `[ { "product_id": 12, "units": 30, "revenue": 1050.00 }, … ]`
    - By day: `[ { "date":"2025-08-18", "units": 52, "revenue": 1780.00 }, … ]`

🔐 Middleware Placement (quick reminder)

- `app.use('/auth', authRouter)` → **no JWT** required.
- `app.use(verifyToken)` → **after** `/auth`, **before** the rest.
- `app.use('/customers', customersRouter)`
- `app.use('/inventory', inventoryRouter)`
- `app.use('/sales', salesRouter)`

🧰 Minimal Body Shapes (students can Zod-validate)

- **Customer (create/update):**

```
{ "first_name": "Ana", "last_name": "Lopez", "email": "ana@ex.com", "pho
ne": "+504…" }
```

- **Product (create/update):**

```
{ "name": "Milk 1L", "price": 35.00, "stock_quantity": 100 }
```

- **Sale (create):**

```
{ "customer_id": 5, "product_id": 12, "quantity": 3, "sale_date": "2025-08-1
```

```
8" }
```
