Final Report of InventoryDB

Database Final Project of Group 2

Developed and written by **Group 2**:

Dimas Stevano Ethanuel	24/532938/PA/22555
Ryan Ethan Halim	24/536718/PA/22765
Hamzah Rajwaa Abdillah	24/546622/PA/23207

Under the supervision of:
Edi Winarko, M.Sc., Ph.D.

DEPARTMENT OF COMPUTER SCIENCE AND ELECTRONICS
FACULTY OF MATHEMATICS AND NATURAL SCIENCES
UNIVERSITAS GADJAH MADA

Special Regency of Yogyakarta
2025

1 Introduction

The lack of a centralized database system makes it difficult for small online businesses to manage their inventory and sales records. Many businesses might use spreadsheets or manual record-keeping, both of which are vulnerable to human error, oversights, redundancy, and inconsistent data. As businesses grow, so do the number of transactions they make, keeping these methods of record-keeping would be inefficient and difficult to manage, ultimately becoming unreliable.

Without an integrated database system, businesses would struggle to maintain accurate and up-to-date records. They might face issues such as false inventory counts, delayed transactions, missing records, and a risk of creating weak analytic reports. Expanding their operations will be a challenge if businesses continue working with manual record-keeping methods.

A reliable and structured database system that supports inventory tracking and sales management would ease the difficulties that may come. Such systems must be capable of centralizing business records, maintaining consistency, automating transaction processes, and providing analytical reports. By implementing an easy to use and well designed database system, small online businesses can improve their operational efficiency and support their long term growth.

Taking the endeavor to aid the tracking and managing inventory and sales respectively of small businesses in a non-convoluted and minimalistic manner, we had set forth a proposal document outlining these objectives:

1) designing a relational database schema which is suitable for the aforementioned tracking of operations;
2) developing a suitable and reliable backend implementation that implements CRUD functionalities constrained in a way which is free of invalid states; and
3) developing a web-based application for user-interface in a manner that is easily accessible for multiple users with varying permissions.

Therefore, with these objectives in mind, this project is developed in the hopes that it may be able to fit in use cases, wherein a centralized small business with multiple administrators may keep track of inventory and sales information.

2 Database Design

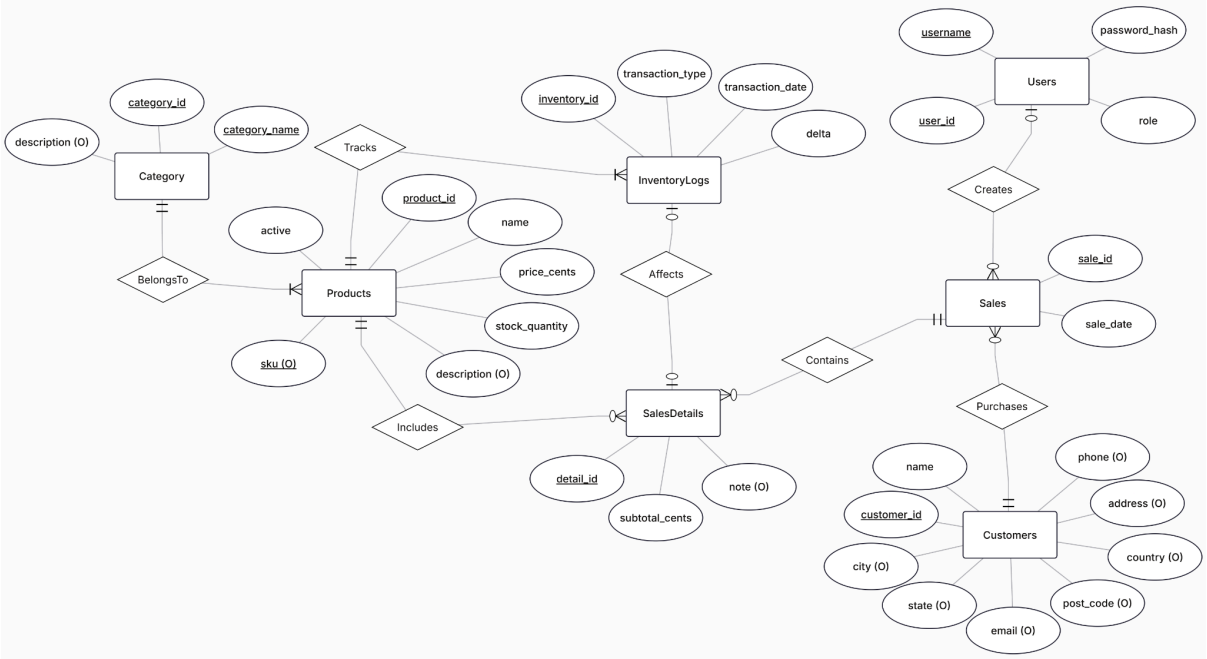

Figure 1. ER diagram of the database schema

Superficially, non-technical wise, there are six entities that exist in the database design. The choices in entities and attributes were made, keeping in mind applying normalization efforts directly. The decisions were based out of minimization and generalization. Therefore, it is already 3NF due to the lack of repetition, partial keys, as well as transitive dependencies by initial design.

2.1 Entities

1) **Users** represent users of the inventory management system.
    a) They are identified through the surrogate primary key **user_id**.
    b) Each user has a unique **username**.
    c) They authenticate using passwords whose **password_hash** are verified.
    d) Each has a **role** which dictates their permissions in the system.
        i) **‘r’ead** may only read the content of the database.
        ii) **‘w’rite** may write to the database, but not delete.
        iii) **‘a’dmin** may do everything, including adding new users.
        iv) **‘d’eactivated** users represent users whose access is revoked.

2) **Customers** represent customers that transact through the system.
    a) They are identified through the surrogate primary key **customer_id**.
    b) They are legally identified through their **name**.
    c) They optionally have address information: street **address**, **city**, **state**, **post_code**, as well as **country**.

d) They optionally have contact information: **phone** number and **email** address.

3) **Categories** represent product categories.
    a) They are identified through the surrogate primary key **category_id**.
    b) Each category has a unique **category_name**.
    c) Each can optionally have a textual **description**.

4) **Products** represent transactable product types.
    a) They are identified through the surrogate primary key **product_id**.
    b) Each product type may optionally have a unique **sku**.
    c) Each product type may be **active** or inactive.
    d) Each product type has a **name** and **price** in **cents**. This will be stored as an integer, because SQLite does not have a decimal type nor can floating point types be relied on for monetary arithmetic operations.
    e) Each product type has a **stock_quantity**.
    f) Each product type can optionally have a **description**.

5) **Sales** represent transactions made by a user at a certain time.
    a) They are identified through the surrogate primary key **sale_id**.
    b) A sale is made at a certain **sale_date**.

6) **SalesDetails** represent each product transaction in a **Sales** entry.
    a) They are identified through the surrogate primary key **detail_id**.
    b) They contain the **subtotal** in **cents**. This does not necessarily equate to multiplying the product's current price by the quantity due to potential historical fluctuations.
    c) Each may optionally have a textual **note** (specifications, discounts, et cetera).
    d) A detail entry does **not** contain the quantity of the product bought, because it is tied to an inventory log.

7) **InventoryLogs** represent logs of changes in product stocks.
    a) They are identified through the surrogate primary key **inventory_id**.
    b) Each log has a **log_type**, signalling its log cause, be it:
        i) **'s'ale** transaction (reduction in quantity);
        ii) **re'f'ill** or restocking (increase in quantity);
        iii) **'r'eturning** of a product (increase in quantity);
        iv) **'d'amage** of a product (reduction in quantity);
        v) **'a'djustment** in quantity (variable); or
        vi) **'o'ther** causes.
    c) Each log is made at a certain **log_date**.
    d) Each log hints at a **delta** change of the quantity of a certain product type.

2.2 Relations

1) **BelongsTo** (**Products** [many] → **Categories** [one])
   Each product optionally **belongs to** one category.

2) **Creates** (**Users** [one] → **Sales** [one])
   Each sale is **created** by some single user.

3) **Purchases** (**Customers** [many] → **Sales** [one])
   Each sale is **purchased** by some single customer.

4) **Contains** (**Sales** [one] → **SalesDetails** [many])
   Each sales detail is linked to some single sale. Each sale **contains** some number of sale details.

5) **Includes** (**SalesDetails** [many] → **Products** [one])
   Each sale detail **includes** some single product.

6) **Affects** (**SalesDetails** [one] → **InventoryLogs** [one])
   Each sale detail **affects** some single inventory log.

7) **Tracks** (**InventoryLogs** [many] → **Products** [one])
   Each inventory log **tracks** a change in some single product.

3 Database Implementation

The DMS used in this project is SQLite, which offers a simple unified method to programmatically interact with the backend, which in this case, is written using Python with the Flask framework and Bcrypt to provide user authentication.

3.1 Logical Diagram and Schema

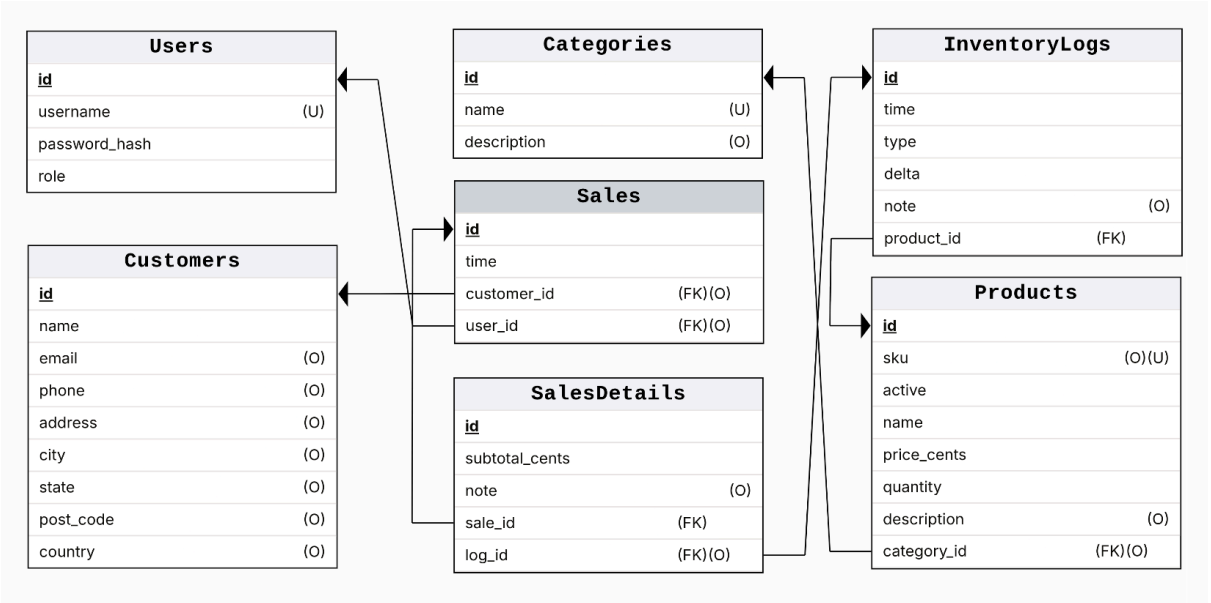

Figure 2. Logical diagram of the underlying schema

The ERD was derived into the corresponding relational structure which is represented by the logical database schema, as shown above. Tables, primary keys, and foreign key relationships that guarantee data consistency and integrity are defined. The schema serves as the basis for the physical database and makes sure that relationships between entities are appropriately maintained by converting the conceptual design into an implementable form, as concretely shown in the SQL file.

3.2 Applying Constraints

Some constraints were warranted, in order to preserve the integrity of the database. These were applied provisionally, as cited below in the snippets. One essential thing to be mentioned is that SQLite only checks for foreign key violations when declared separately as explicit constraints. Without repeating explanation, by default, when foreign keys are updated or deleted, they trigger cascading updates as well as reference-deletions (set-null) respectively.

```
CREATE TABLE IF NOT EXISTS users (
```

```
    ...

    CONSTRAINT username_check UNIQUE (username),
    CONSTRAINT role_check CHECK (role IN ('d', 'r', 'w', 'a'))
    -- `d`eactivated (None), `r`ead (R), `w`rite (CR), `a`dmin (CRUD)
);
```

*Snippet 1. Constraints for the **Users** table to ensure username uniqueness and role validity*

```
CREATE TABLE IF NOT EXISTS customers (
    ...

    CONSTRAINT email_check CHECK (email LIKE '%_@__%.__%')
);
```

*Snippet 2. Constraint for the **Customers** table to ensure e-mail address validity*

```
CREATE TABLE IF NOT EXISTS categories (
    ...

    CONSTRAINT name_check UNIQUE (name)
);
```

*Snippet 3. Constraint for the **Categories** table to ensure name uniqueness*

```
CREATE TABLE IF NOT EXISTS products (
    ...

    CONSTRAINT sku_check UNIQUE (sku),
    CONSTRAINT active_check CHECK (active IN (0, 1)),
    CONSTRAINT price_check CHECK (price_cents > 0),
    CONSTRAINT quantity_check CHECK (quantity >= 0),
    CONSTRAINT category_id_check FOREIGN KEY (category_id) REFERENCES categories(id)
        ON UPDATE CASCADE ON DELETE SET NULL
);
```

*Snippet 4. Constraints for the **Products** table to ensure SKU uniqueness, type correctness[1], and foreign key update & deletion behaviors to their respective categories*

[1] Unlike MySQL, SQLite does not have unsigned integer nor boolean types to accommodate for **active_check**, **price_check**, and **quantity_check**. Thus, these constraints must be applied directly.

```sql
CREATE TABLE IF NOT EXISTS inventory_logs (
   ...

   CONSTRAINT type_check CHECK (type IN ('s', 'f', 'r', 'd', 'a', 'o')),
   -- `s`ale, re`f`ill (restock), `r`eturned, `d`amaged, `a`djustment, `o`ther
   CONSTRAINT product_id_check FOREIGN KEY (product_id) REFERENCES products(id)
       ON UPDATE CASCADE ON DELETE SET NULL
);
```

*Snippet 5. Constraints for the **InventoryLogs** table to ensure log type validity, as well as foreign key update & deletion behaviors to their respective products*

```sql
CREATE TABLE IF NOT EXISTS sales (
   ...

   CONSTRAINT customer_id_check FOREIGN KEY (customer_id) REFERENCES customers(id)
       ON UPDATE CASCADE ON DELETE SET NULL,
   CONSTRAINT user_id_check FOREIGN KEY (user_id) REFERENCES users(id)
       ON UPDATE CASCADE ON DELETE SET NULL
);
```

*Snippet 6. Constraints for the **Sales** table to ensure foreign key update & deletion behaviors to their respective buying customers as well as logging users*

```sql
CREATE TABLE IF NOT EXISTS sales_details (
   ...

   CONSTRAINT sale_id_check FOREIGN KEY (sale_id) REFERENCES sales(id)
       ON UPDATE CASCADE ON DELETE CASCADE,
   CONSTRAINT log_id_check_unique UNIQUE (log_id),
   CONSTRAINT log_id_check FOREIGN KEY (log_id) REFERENCES inventory_logs(id)
       ON UPDATE CASCADE ON DELETE SET NULL
);
```

*Snippet 7. Constraints for the **SalesDetails** table to ensure the uniqueness of the* referenced *__InventoryLogs__ entry, as well as foreign key update & deletion behaviors*

3.3 Applying Triggers

SQLite does not precompute, cache, nor maximally optimize queries. Therefore, while it is possible to compute the quantity of a product type by using an aggregate query of all inventory log entries pertaining to the specified product, like

exemplified below, this approach is wasteful and nonoptimal, especially as it is an O(n) operation and would have to be called for every product in the user-interface.

```
SELECT sku, SUM(delta) AS quantity
FROM products p LEFT JOIN inventory_logs il ON p.id = il.product_id
GROUP BY products.id;
```

Snippet 8. A query which may have an $O(n \times m)$ time complexity every invocation

Thus, every time an inventory log entry is inserted, deleted, or updated, triggers are relied upon to automatically update the **quantity** attribute of **InventoryLogs**, providing a constant-time access to the product quantities.

```
CREATE TRIGGER IF NOT EXISTS insert_quantity_delta
AFTER INSERT ON inventory_logs
BEGIN
   UPDATE products
   SET quantity = products.quantity + NEW.delta
   WHERE products.id = NEW.product_id;
END;
```

Snippet 9. Insertion trigger to change the quantity by the inserted delta

```
CREATE TRIGGER IF NOT EXISTS delete_quantity_delta
AFTER DELETE ON inventory_logs
BEGIN
   UPDATE products
   SET quantity = products.quantity - OLD.delta
   WHERE products.id = OLD.product_id;
END;
```

Snippet 10. Deletion trigger to undo the quantity by the delta change

```
CREATE TRIGGER IF NOT EXISTS update_quantity_delta
AFTER UPDATE ON inventory_logs
BEGIN
   UPDATE products
   SET quantity = products.quantity - OLD.delta
   WHERE products.id = OLD.product_id;

   UPDATE products
   SET quantity = products.quantity + NEW.delta
```

```
    WHERE products.id = NEW.product_id;
END;
```

Snippet 11. Update trigger to undo by the old delta change and redo by the new delta

For the same reason, these triggers are used to set the **total_cents** attribute of the **Sales** table to the sum of all of its associated **SalesDetails**' **subtotal_cents** attribute, similarly written without any substantial changes.

3.4 Backend Implementation

The backend provides access to the SQLite database over the network, which includes all CRUD operations (**create**, **read**, **update**, and **delete**). The layer, implementing a REST API (exchanging JSONs), especially handles user authentication, permission enforcement, query validation, as well as opaque query calling (interaction with the SQLite library). The implementation can be found here.

API Route	REST Methods	Correspondence
/api/me	GET	Authentication: get current user
/api/login	POST*	Authentication: log in as a user with password
/api/logout	POST*	Authentication: log out the session
/api/products	GET, POST*	Everything from the **Products** table
/api/products/**{id}**	GET, PUT*, DELETE*	Information of a **Products** entry with the **{id}**
/api/users	GET, POST*	Everything from the **Users** table
/api/users/**{id}**	GET, PUT*, DELETE*	Information of a **Users** entry with the **{id}**
/api/categories	GET, POST*	Everything from the **Categories** table
/api/categories/**{id}**	GET, PUT*, DELETE*	Information of a **Categories** entry with the **{id}**
/api/customers	GET, POST*	Everything from the **Customers** table
/api/customers/**{id}**	GET, PUT*, DELETE*	Information of a **Customers** entry with the **{id}**
/api/sales	GET, POST*	Everything from the **Sales** table
/api/sales/**{id}**	GET, DELETE*	**SalesDetails** of a **Sales** entry with the **{id}**
/api/logs	GET, POST*	Everything from the **InventoryLogs** table
/api/logs/**{id}**	GET, PUT*, DELETE*	Information of a **InventoryLogs** entry with the **{id}**

*) Unprivileged operations *) Write operations *) Admin operations

4 Application Implementation

For the frontend, we decided to build a **Single Page Application (SPA)**. Instead of relying on heavy frameworks like React or Vue, we stuck to vanilla web technologies such as HTML5, CSS3, and modern JavaScript. We wanted the application to be lightweight and fast, and building it from scratch gave us complete control over how everything works under the hood. It feels snappy because the page never actually reloads; it just swaps out the content dynamically as you navigate.

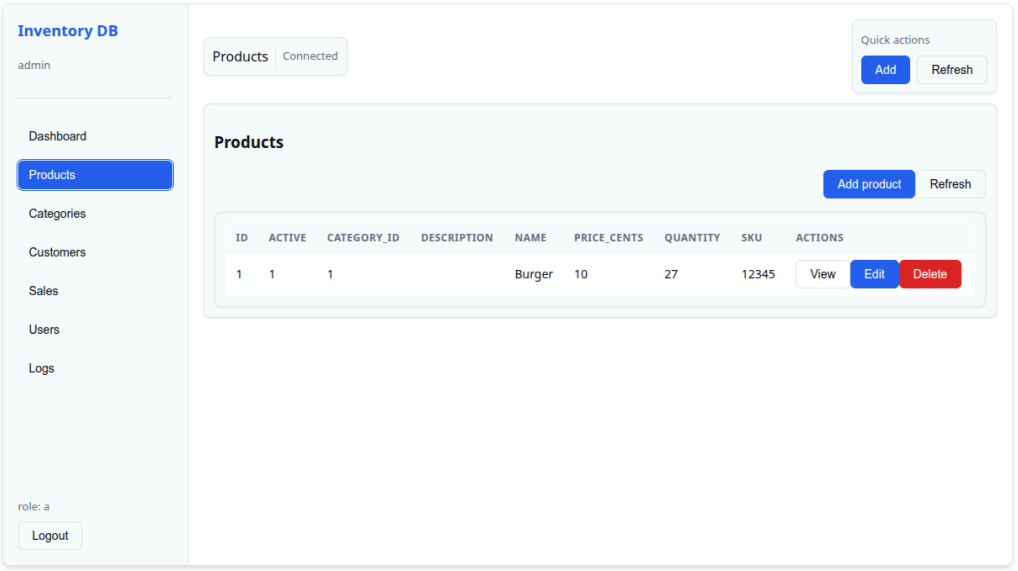

Image 1. Overall layout of the frontend

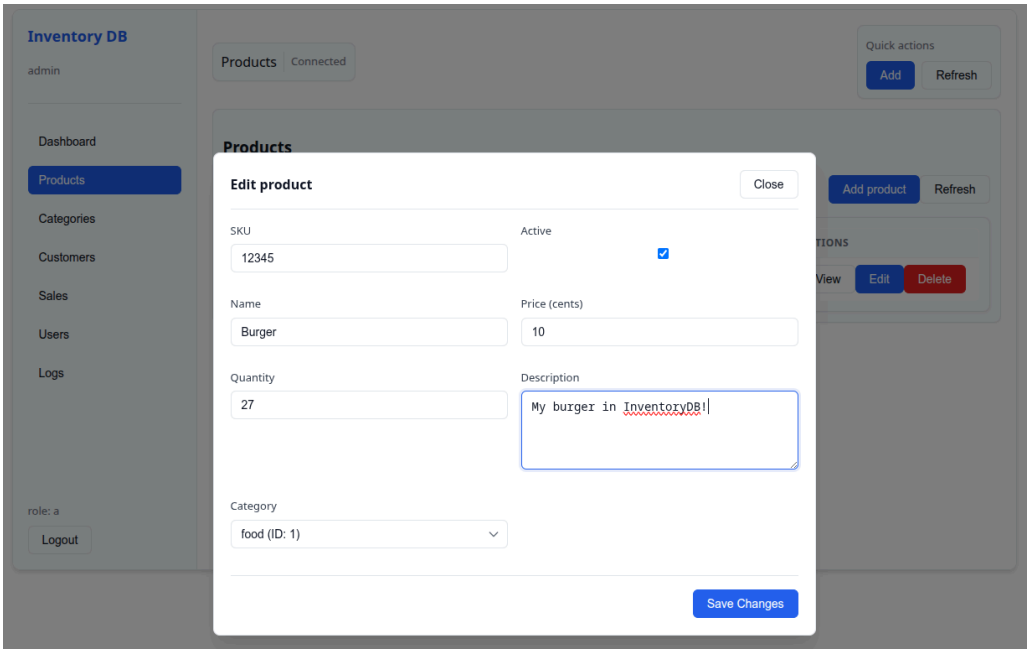

Image 2. Modal window in editing and inserting entries

4.1 Architecture & Structure

The whole application lives inside a single file, **index.html**.

- Single Entry Point: The index.html file contains the skeleton for both the login view and the Main Application Layout. These views are toggled using JavaScript based on the user's authentication state.
- Component-Based Layout:
    - Sidebar: Provides persistent navigation to different modules (Dashboard, Products, Sales, etc.).
    - Topbar: Displays the current view title, connection status, and quick actions.
    - Content Area: A dynamic section where data tables, forms, and statistics are rendered.
    - Modal System: A dedicated root element (**#modal-root**) is used to render overlays for creating or editing records, ensuring they sit above the main content.

4.2 User Interface (UI) Design

We did all the styling in **style.css**. We wanted the design to be clean and easy to maintain, so we used CSS Variables. We defined our colors (like the accent blue, background grays, and alert reds) in one place. This means if we ever want to change the theme or add a "Dark Mode" later, we only have to tweak a few lines of code.

We used **Flexbox** for almost everything. It made laying out the sidebar and the main content area really intuitive, and it ensures the app looks good whether you're on a small laptop screen or a large desktop monitor.

4.3 The Brains of the Operation (JavaScript)

The real magic happens in **script.js.** This file acts as the brain of the frontend.

- **Managing State**: We created a global **STATE** object to keep track of what's happening right now—who is logged in, what page they are looking at, and the authentication token.
- **Talking to the Server:** We wrote a helper function called **apiFetch()**. Instead of writing out the full **fetch** code every time we need data, we just call this function. It automatically handles the boring stuff like adding the security token to the header and checking for errors.
- **Building the Page:** Since we're not using a framework, we wrote a simple rendering engine. When you click "Products," the code fetches the list from the server and loops through it, building the HTML table row-by-row in real-time.

4.4 Key Features

- **Security First:** The app checks if you're logged in the moment it loads. If you aren't, it boots you straight to the login screen.

- **Role-Based Access:** We made the UI smart enough to know who you are. If you're an admin, you see everything. If you're a standard user, the app automatically hides sensitive buttons (like "Delete User") so you can't accidentally break anything.
- **Dynamic Forms:** Whenever you need to add data, like a new Sale, the app opens a modal. We wrote the code to handle these form submissions in the background (asynchronously), so the user never has to wait for a page reload to see their changes.

5 Testing and Results

5.1 Backend API

5.1.1 Logging In

By default, the backend creates a default **admin** user with the password **admin**.

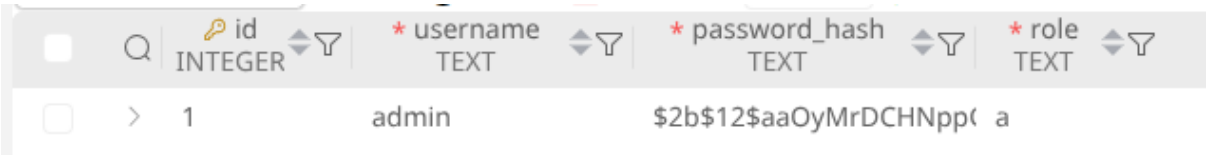

*Image 3. Preview of the internal **Users** table*

Thus, firstly, we log in as the admin with full permissions by using a **POST** request:

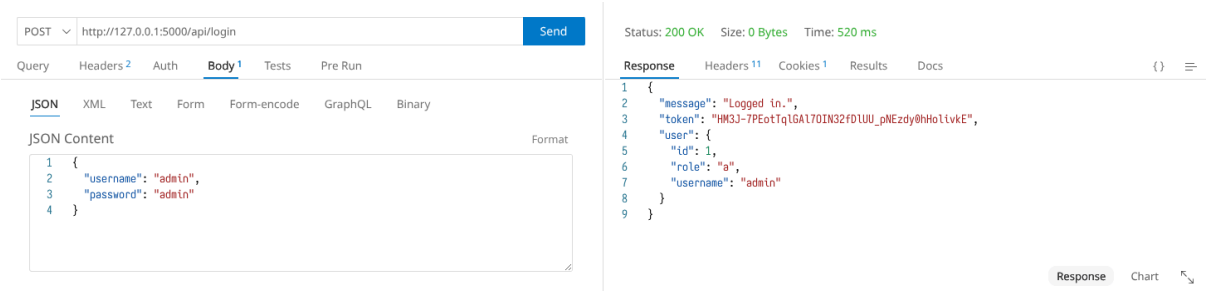

Image 4. Pushing a POST request to /api/login with the JSON credentials

5.1.2 Create (POST)

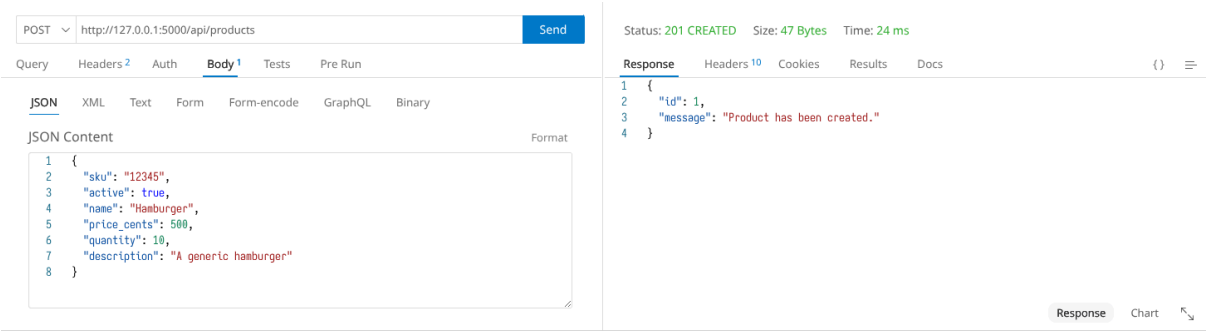

Image 5. Pushing a POST request to /api/products with the product to be added whose attributes are placed into the JSON

5.1.3 Read (GET)

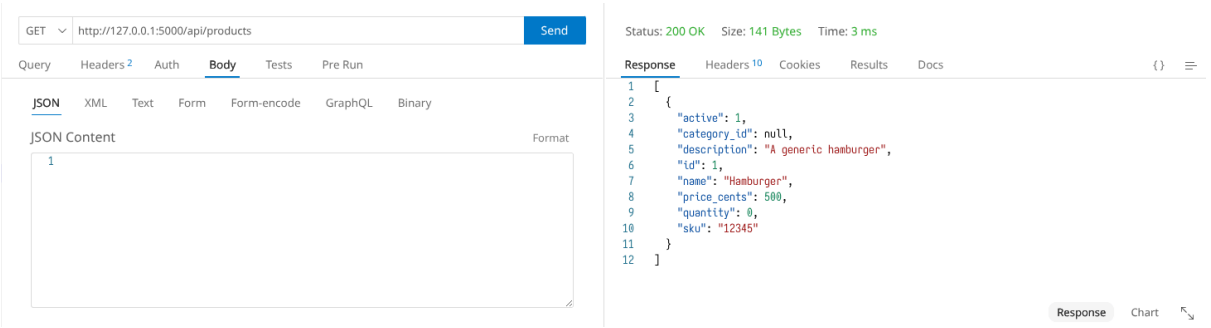

*Image 6. Pushing a GET request to /api/products, in order to get a list of all entries in the **Products** table, notably the one we created*

5.1.4 Update

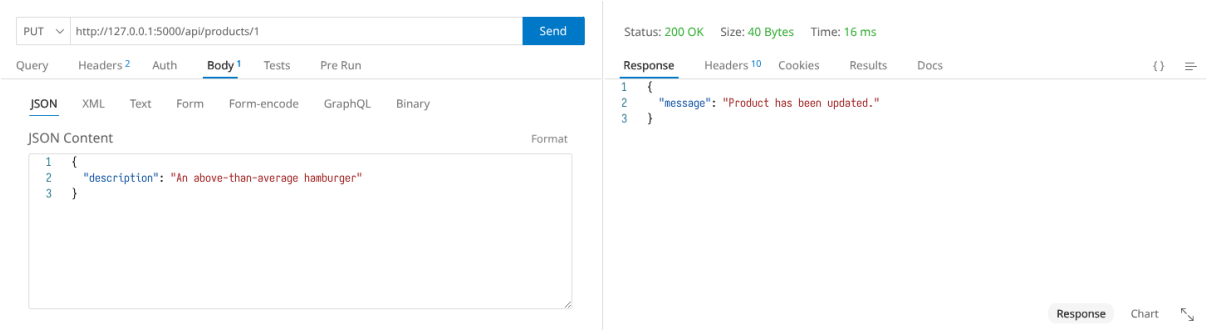

*Image 7. Pushing a PUT request to /api/products/**{id}** with the previously added product of ID: 1 to modify its description*

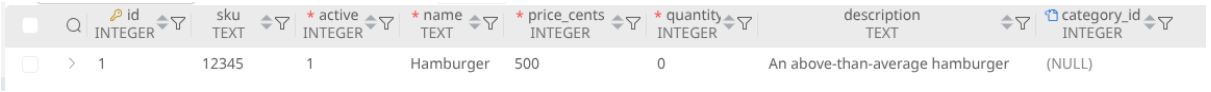

*Image 8. Preview of the internal **Products** table following the update*

5.1.5 Delete

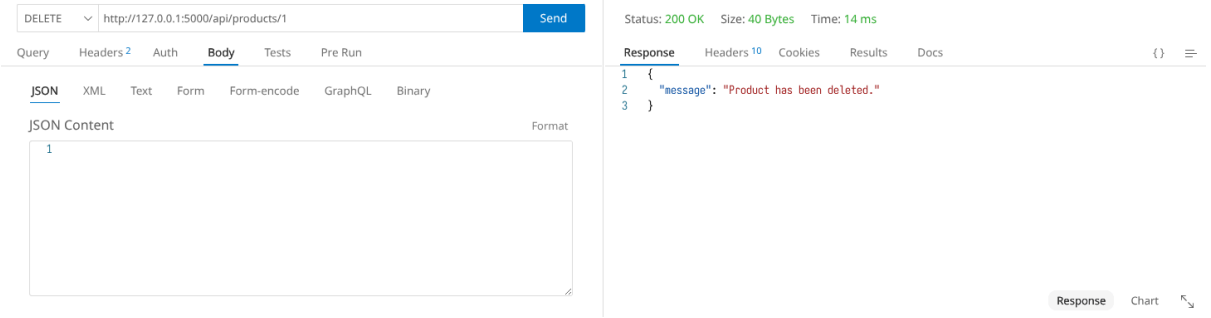

*Image 9. Pushing a DELETE request to /api/products/**{id}** with the previously added product of ID: 1 to remove it*

*Image 10. Preview of the empty internal **Products** table following the deletion*

5.1.6 Logging Out

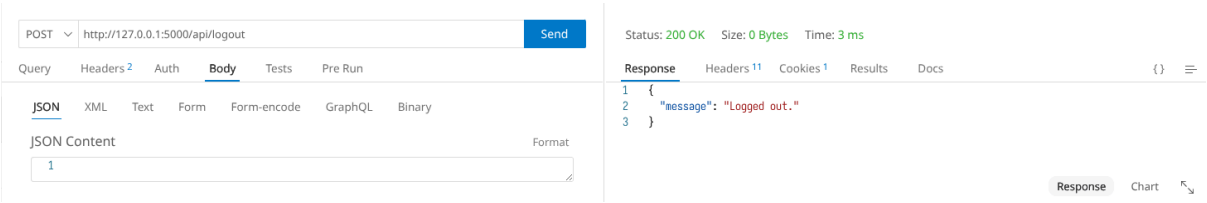

Image 11. Pushing a POST request to /api/logout to clear the session

5.2 Frontend

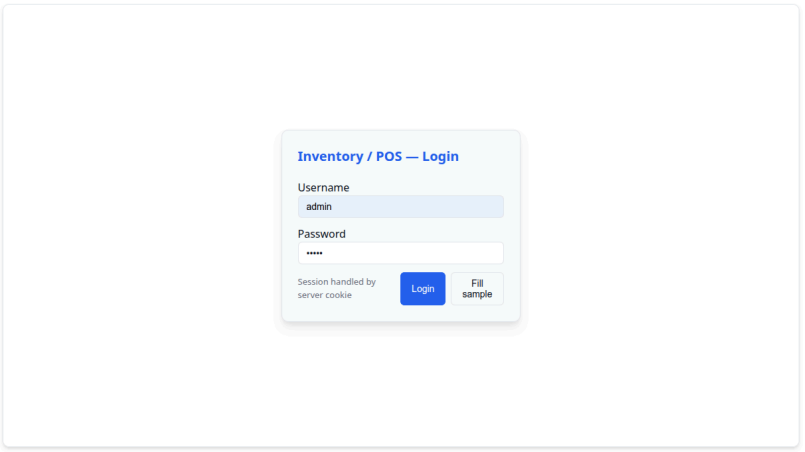

Image 12. Login page

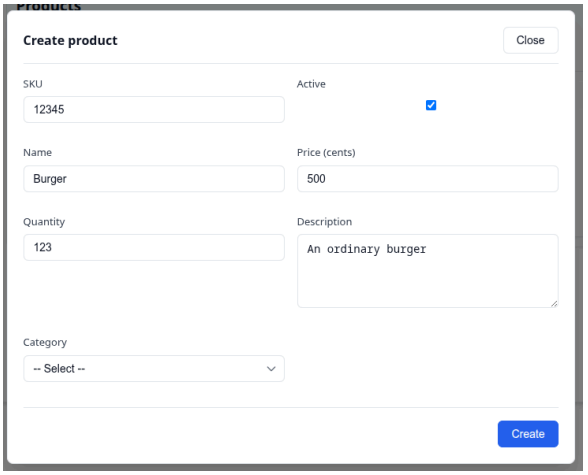

Image 13. Product creation

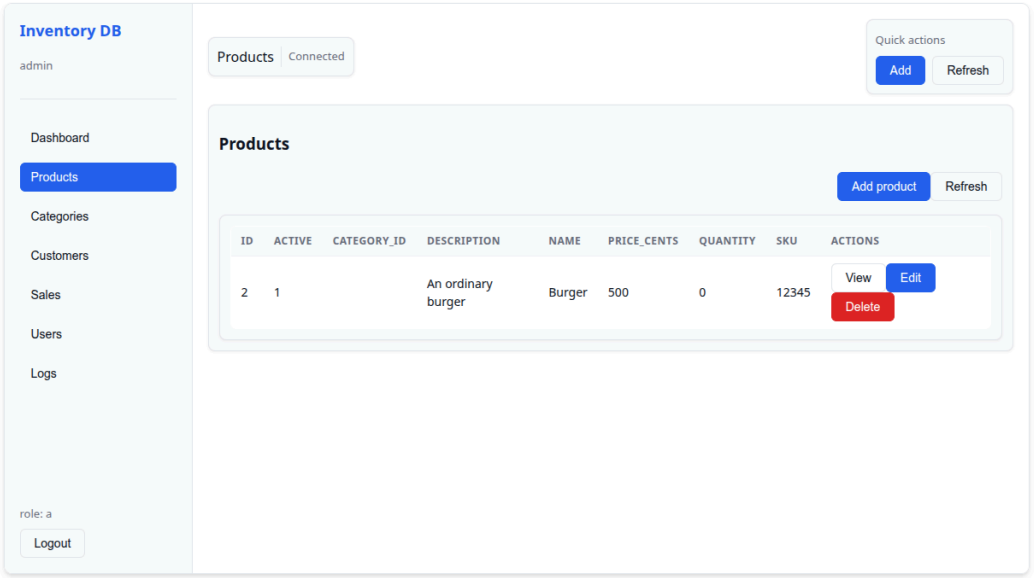

Image 14. Preview of all products

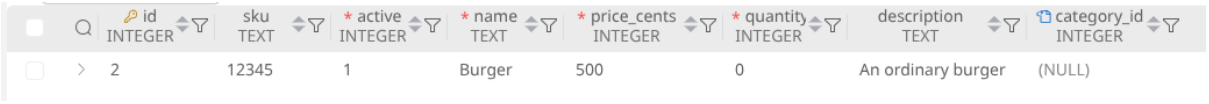

*Image 15. Preview of the internal **Products** table following the creation*

6 Conclusion and Reflection

InventoryDB, the inventory management system developed in this project successfully demonstrates a full-stack web application capable of handling essential business operations. By integrating a Python Flask backend with a robust SQLite database and a responsive frontend, the system provides a centralized solution for tracking products, managing customers, and recording sales, intended for small businesses. The system in particular has the possibility of becoming fruitful in providing aid thereto. The final system allows users to manage products, record sales, track inventory changes, as well as store customer information in a centralized manner.

We undertook the process of designing, developing, and putting databases into use for a real life scenario. One important lesson point that can be taken is the fundamental importance of careful database design at the start before any code is written. Databases optimally must be designed in such a way that ideally provides simplicity, avoids duplication or redundant information (undergoing normalization), as well as maximizing performance. We learned by using SQLite that not all DMS are equal, and that we had to think differently in terms of database logic, especially when we had to manually deal with the constraints due to the lack of nuanced types (like ones provided in MySQL) and had to make workarounds using triggers and “precomputed” attributes. It is rather understated that denormalization is important in cases like these.

The most challenging part was the architecture pipelining. There is so much work that has to be duplicated over from the database design, backend development, as well as frontend development, because each attribute name ought to be duplicated and verified of its integrity early on along the way. This leads to multiple points of failure in development, since any problem we might encounter may come from either the frontend, the backend, or the database schema instead. This isn’t helped by the fact that one can’t simply change the schema of a database due to existing data that already occupies the schema. Thus, one challenge in production that we might encounter would be data migration.

In terms of improvements, we believe that the backend could’ve been made to do periodic backups in cases of power failure, where SQLite wouldn’t be able to rollback changes, compromising the ACID principle. As for the frontend, unfortunately, due to the skill insufficiencies and technical hurdles on our part, we weren’t able to make it as interactive and providing of insightful information as we initially hoped. Therefore, quality-of-life features, such as monthly report in revenues, graphs to visualize inventory changes of a product, et cetera, would have been warranted.