Final report: Inventory management system

Group Members

1. Nguyen Xuan Truong
2. Le Ngoc toan

Contents

1 Conceptual & Logical Design

1.1 Functional Requirements

These specify what the system must do, focusing on the capabilities required for the inventory and order management system.

User Management

- **Create/Update/Delete Users:** Administrators can manage user accounts stored in the `users` table, which includes fields such as `username`, `email`, `password_hash`, `role_name` (`admin` or `staff`), and `warehouse_id`.
- **Authentication:** Users must log in using their email and password. Login attempts are recorded in the `login_logs` table, including `ip_address`, `user_agent`, and `refresh_token`.
- **Role-Based Access:** Admins have full access. Staff users are limited to managing products and orders within their assigned `warehouse_id`.

  Example: An admin creates a user with `role_name = 'staff'` and assigns them to `warehouse_id = 1`. The system logs their login on 2025-05-20 22:17:00 from IP 192.168.1.1.

Warehouse Management

- **Manage Warehouses:** Warehouses are stored and updated in the `warehouses` table with fields like `name` and `address`.
- **Associate Entities:** Warehouses are linked to both users and products via `warehouse_id`.

  Example: A warehouse named "Central Hub" at "123 Main St" is linked to multiple products and users.

Supplier Management

- **Manage Suppliers:** Supplier data is stored in the `suppliers` table, including `name`, `contact_name`, `contact_email`, and `phone`.
- **Link to Products:** Each product is linked to a supplier via `supplier_id`.

  Example: A supplier "TechCorp" with email "contact@techcorp.com" supplies products like "Laptop" and "Mouse".

Category Management

- **Manage Categories:** Categories are stored in the `categories` table with fields `name` and `description`.
- **Link to Products:** Products reference a `category_id` to indicate category.

  Example: A category "Electronics" with description "Gadgets and devices" includes products like "Smartphone".

Product Management

- **Manage Products:** Products are stored in the `products` table with fields `name`, `description`, `price`, `quantity`, and `image_url`, linked to `suppliers`, `warehouses`, and `categories`.
- **Inventory Tracking:** The system tracks quantity per warehouse and updates it when orders are placed.

  Example: A product "Laptop" with price 999.99 and quantity 50 is stored in warehouse "Central Hub" and belongs to the "Electronics" category.

Customer Management

- **Manage Customers:** Customers are stored in the `customers` table with fields such as `name`, `email`, `phone`, and `address`.

  Example: Customer "John Doe" with email "john@example.com" has placed multiple orders.

Order Management

- **Create/Track Orders:** Orders are stored in the `orders` table, including `customer_id`, `order_date`, and `status` (`pending`, `completed`, `cancelled`).
- **Link to Order Items:** Orders are connected to `order_items`, which link to `products` through `product_order_items`.

  Example: Order with `order_id = 1` for customer "John Doe" on 2025-05-20 includes two laptops (quantity = 2, total_price = 1999.98).

Order Item Tracking

- **Manage Order Items:** Order items are tracked in the `order_items` table and linked to products via `product_order_items` including `quantity` and `total_price`.

  Example: `order_item_id = 1` links to `product_id = 1` (Laptop) with quantity = 2.

1.2 Non-Functional Requirements

These define system qualities like performance, security, and maintainability.

- **Performance:** Queries (e.g., product searches, order lookups) should return results in under 2 seconds for up to 50 concurrent users.

  Example: A product search by category uses an index on `products.category_id` for fast retrieval.
- **Scalability:** The system must support a modest scale of up to a few thousand products and hundreds of orders without significant performance degradation.

  Example: Partitioning orders by `order_date` ensures efficient access.
- **Security:** Passwords must be hashed (e.g., bcrypt). All login attempts must be logged in `login_logs` to detect suspicious access.

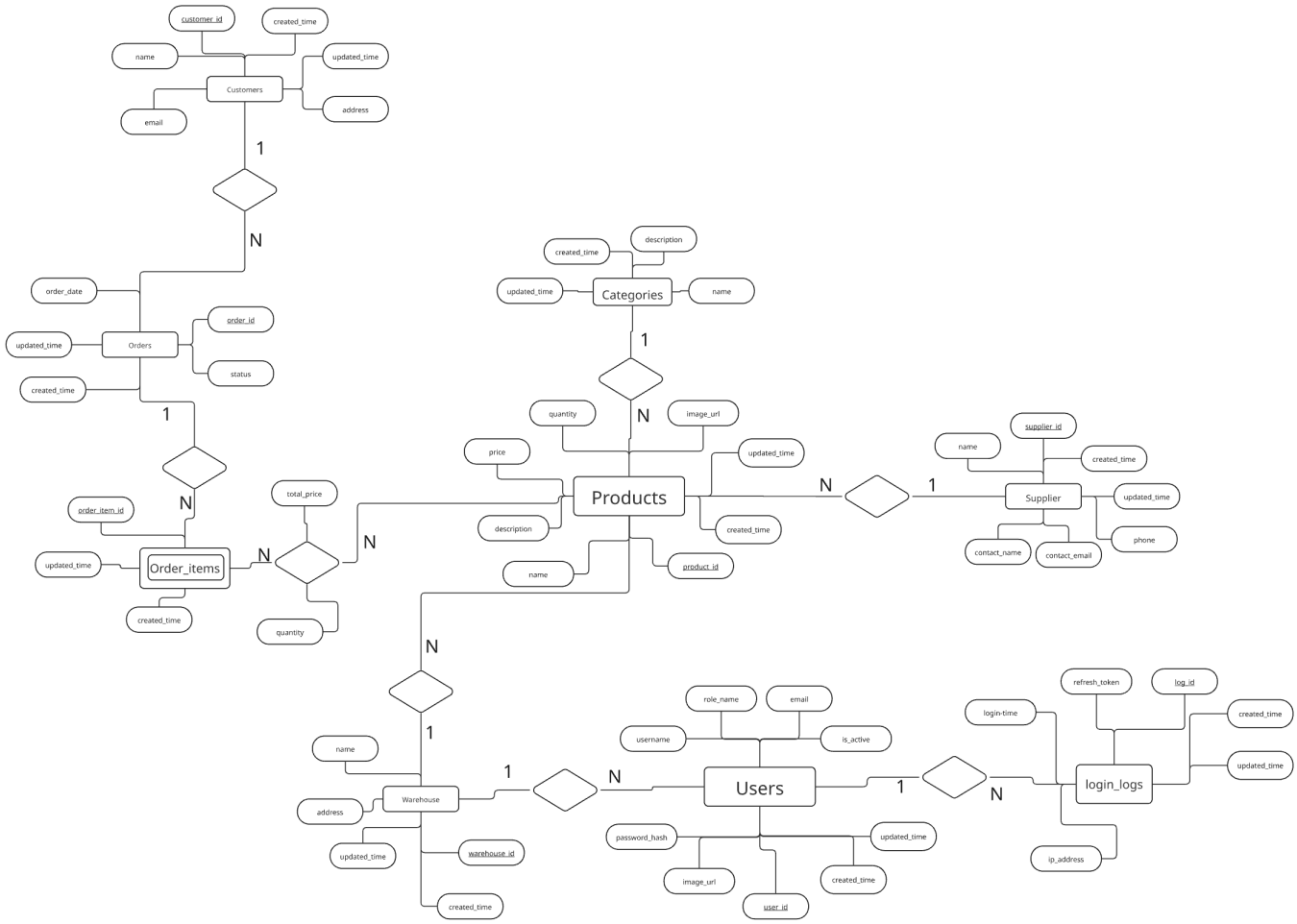
  Example: A failed login attempt for `user_id = 1` is recorded with the `ip_address` and `user_agent`.
- **Reliability:** Foreign key constraints (e.g., `fk_products_supplier`) ensure data integrity. Order updates must be atomic and isolated to prevent race conditions.

  Example: Updating `products.quantity` during an order is handled in a transaction.
- **Availability:** The system should maintain 99.5% uptime. A simple master-slave database setup ensures basic failover support.

  Example: A replicated MySQL setup improves resilience during maintenance.
- **Maintainability:** The schema must be fully normalized (to at least 3NF) to reduce redundancy and simplify modifications.

  Example: Adding a new attribute to a product only requires a column addition in the `products` table.
- **Architecture and Technologies:**
  - Fully normalized schema (3NF)
  - Fast API responses with caching (e.g., Redis)
  - Secure authentication using JWT and bcrypt-hashed passwords
  - ACID-compliant database transactions
  - Indexed queries for optimized performance
  - Modular, containerized architecture using Docker
  - Responsive and intuitive UI (e.g., built with Next.js)
  - CI/CD-enabled development workflow

1.3 Entity–Relationship Diagram



- **Warehouses** have a one-to-many relationship with:
  - **Users**
  - **Products**
- **Suppliers** have a one-to-many relationship with **Products**.
- **Categories** have a one-to-many relationship with **Products**.
- **Customers** place multiple **Orders** (one-to-many).
- **Orders** contain multiple **Order_Items** (one-to-many).
- **Products** and **Order_Items** have a many-to-many relationship through the **Product_Order_Items** junction table.

1.4 Normalization proof up to Third Normal Form (3NF)

This section provides a detailed normalization proof for the database schema, demonstrating that all tables conform to the rules of First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF).

First Normal Form (1NF)

Requirement: All attributes must be atomic (no multi-valued attributes or repeating groups), and each table must have a primary key.

Analysis:

- **Atomic Attributes:** Every attribute in all tables contains only atomic values. For instance:
  - `users.email` stores one unique email address per user.
  - `products.price` stores a single decimal value.
  - `customers.address` uses a `TEXT` field but holds a single address per customer, not multiple.
  - `products.quantity` is an integer that represents a single count value.
- **Primary Keys:**
  - **Single-column primary keys:** `warehouses(warehouse_id)`, `users(user_id)`, `products(product_id)`, `suppliers(supplier_id)`, `categories(category_id)`, `orders(order_id)`, `order_items(order_item_id)`, etc.
  - **Composite primary key:** `product_order_items(product_id, order_item_id)` uses a composite key to identify the relationship between a product and an order item.

Conclusion: All tables have primary keys and atomic attributes. Therefore, the schema satisfies 1NF.

Second Normal Form (2NF)

Requirement: The schema must first satisfy 1NF, and all non-key attributes must depend on the entire primary key (i.e., no partial dependencies).

Analysis:

- **Tables with single-column primary keys:**
  - Examples include `users`, `products`, `orders`, etc.
  - All non-key attributes in these tables are fully functionally dependent on the primary key. For instance:
    - In `users`, attributes like `email`, `role_name`, and `is_active` depend entirely on `user_id`.
    - In `products`, attributes such as `name`, `price`, `quantity`, etc., depend entirely on `product_id`.
- **Table with composite key:**
  - `product_order_items(product_id, order_item_id)` uses a composite key.
  - Attributes `quantity` and `total_price` depend on the entire composite key.
  - Example: `quantity = 2` only makes sense when referring to a specific combination of `product_id` and `order_item_id`, not just one of them.

Conclusion: There are no partial dependencies; every non-key attribute depends on the full key. Thus, all tables are in 2NF.

Third Normal Form (3NF)

Requirement: The schema must satisfy 2NF and have no transitive dependencies. That is, no non-key attribute should depend on another non-key attribute.

Analysis:

- **users:** All non-key attributes (e.g., `username`, `email`, `is_active`) depend directly on the primary key `user_id`. Although `warehouse_id` is a foreign key, it does not introduce transitive dependency (e.g., `email` does not depend on `warehouse_id`).
- **products:** Attributes such as `name`, `price`, and `quantity` depend only on `product_id`. Foreign keys like `supplier_id` and `category_id` do not cause transitive dependencies. For example, `price` does not depend on `supplier_id`.
- **orders:** Attributes `order_date` and `status` depend on `order_id`, not on `customer_id`. There is no chaining of dependencies among non-key attributes.
- **Avoidance of transitive storage:** If, for instance, `supplier_name` were stored directly in the `products` table, it would be transitively dependent via `supplier_id`. However, the schema correctly avoids this by referencing the name through a foreign key.

Conclusion: No transitive dependencies exist in any table. Therefore, the schema satisfies 3NF.

Final Remark: Since the schema adheres to 1NF, 2NF, and 3NF, it is properly normalized up to **Third Normal Form (3NF)**.

2 Implement Database Entities

2.1 Database Schema

The complete database schema including table definitions, constraints, and relationships can be found in the following GitHub repository:

- https://github.com/truongng201/Database-IMS-Project/tree/main/ims-database/init-scripts

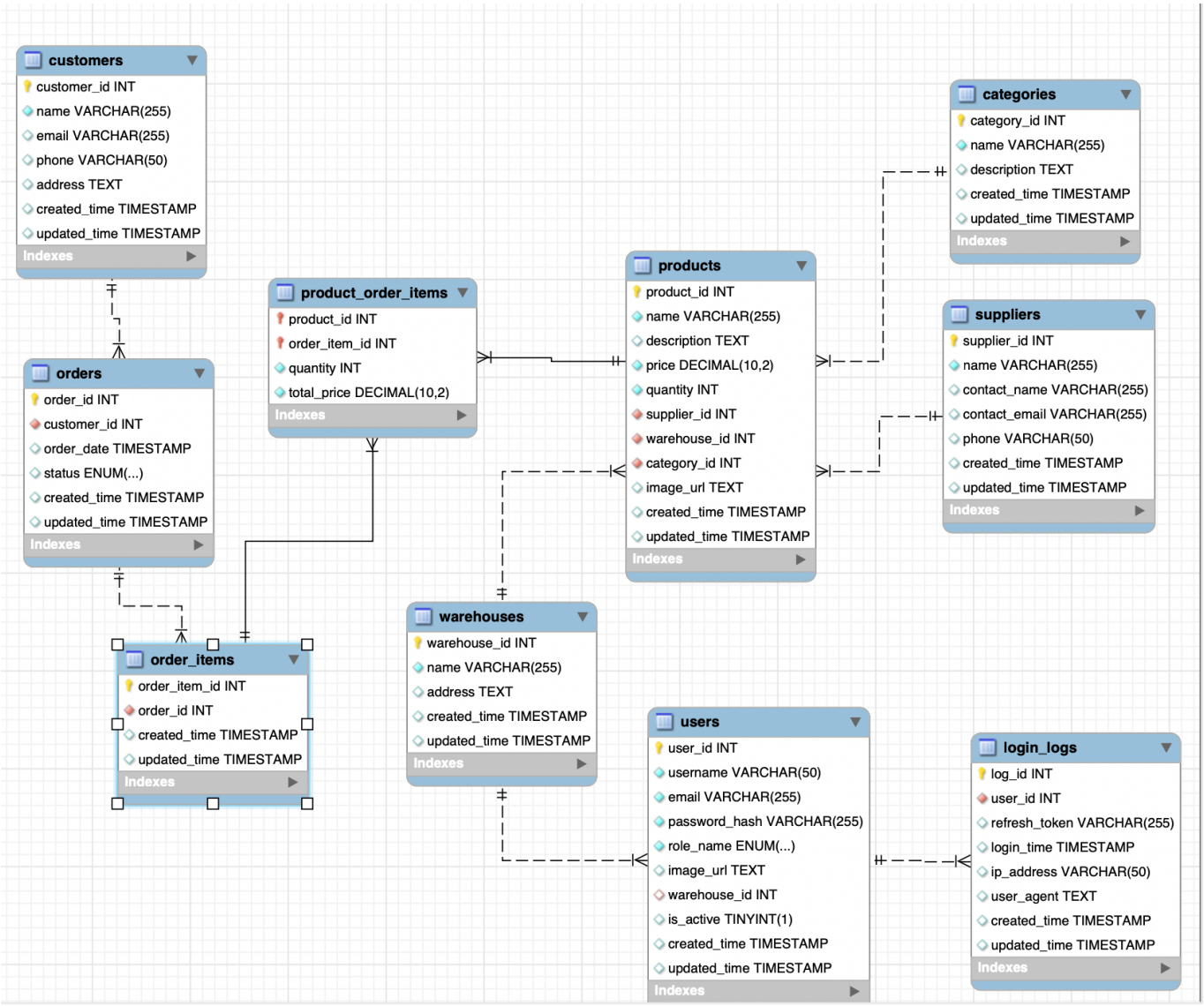


Figure 1: Database Schema Diagram

2.2 Views

The system defines eight views to enhance readability, simplify complex queries, and support user-friendly data summaries for the admin dashboard and analytics.

1. `supplier_products_view`
   - Joins warehouses, suppliers, products, and categories.
   - Provides detailed information about each supplier and the products they provide to each warehouse.

```sql
CREATE OR REPLACE VIEW supplier_products_view AS
SELECT
    w.warehouse_id,
    w.name AS warehouse_name,
    s.supplier_id,
    s.name AS supplier_name,
    s.contact_name,
    s.contact_email,
    s.phone AS contact_phone,
    p.product_id,
    p.name AS product_name,
    p.description,
    p.price,
    p.quantity,
    c.category_id,
    c.name AS category_name,
    p.created_time AS product_created_time,
    p.updated_time AS product_updated_time
FROM
    suppliers s
    LEFT JOIN products p ON s.supplier_id = p.supplier_id
    LEFT JOIN warehouses w ON p.warehouse_id = w.warehouse_id
    LEFT JOIN categories c ON p.category_id = c.category_id;
```

2. `all_suppliers_by_warehouse_view`

- Groups suppliers by warehouse and summarizes total products, quantity, and average price.
- Useful for warehouse-wise supplier insights.

```sql
CREATE OR REPLACE VIEW all_suppliers_by_warehouse_view AS
SELECT
    w.warehouse_id,
    w.name AS warehouse_name,
    s.supplier_id,
    s.name AS supplier_name,
    s.contact_name,
    s.contact_email,
    s.phone,
    COALESCE(COUNT(p.product_id), 0) AS total_products,
    COALESCE(SUM(p.quantity), 0) AS total_product_quantity,
    COALESCE(AVG(p.price), 0) AS avg_product_price,
    MIN(p.created_time) AS earliest_product_created,
    MAX(p.updated_time) AS latest_product_updated,
    s.created_time AS supplier_created_time,
    s.updated_time AS supplier_updated_time
FROM
    suppliers s
    LEFT JOIN products p ON s.supplier_id = p.supplier_id
    LEFT JOIN warehouses w ON p.warehouse_id = w.warehouse_id
GROUP BY
    w.warehouse_id,
    w.name,
    s.supplier_id,
    s.name,
    s.contact_name,
    s.contact_email,
    s.phone,
    s.created_time,
    s.updated_time;
```

3. `customer_summary_view`

- Summarizes each customer's order history, total spending, and most recent purchase.
- Enables quick access to customer value and activity.

```
CREATE OR REPLACE VIEW customer_summary_view AS
SELECT
    c.customer_id AS customer_id,
    c.name AS name,
    c.email AS email,
    c.phone AS phone,
    c.address AS address,
    c.updated_time AS customer_updated_time,
    COUNT(DISTINCT CASE WHEN poi.product_id IS NOT NULL THEN o.order_id END) AS
        total_number_orders,
    COALESCE(SUM(poi.total_price), 0) AS total_spent,
    MAX(CASE WHEN poi.product_id IS NOT NULL THEN o.order_date END) AS last_purchase
FROM
    customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    LEFT JOIN order_items oi ON o.order_id = oi.order_id
    LEFT JOIN product_order_items poi ON oi.order_item_id = poi.order_item_id
GROUP BY
    c.customer_id, c.name, c.email, c.phone, c.updated_time;
```

4. `order_summary_view`

- Summarizes orders including total items, value, and unique products.
- Facilitates administrative monitoring of order trends.

```
CREATE OR REPLACE VIEW order_summary_view AS
SELECT
    o.order_id,
    o.order_date,
    o.status,
    c.customer_id,
    c.name AS customer_name,
    c.email AS customer_email,
    c.phone AS customer_phone,
    p.warehouse_id,
    w.name AS warehouse_name,
    SUM(poi.quantity) AS total_items,
    SUM(poi.total_price) AS total_order_value,
    COUNT(DISTINCT poi.product_id) AS unique_products_ordered
FROM
    orders o
    LEFT JOIN customers c ON o.customer_id = c.customer_id
    LEFT JOIN order_items oi ON o.order_id = oi.order_id
    LEFT JOIN product_order_items poi ON oi.order_item_id = poi.order_item_id
    LEFT JOIN products p ON poi.product_id = p.product_id
    LEFT JOIN warehouses w ON p.warehouse_id = w.warehouse_id
WHERE
    p.product_id IS NOT NULL
GROUP BY
    o.order_id,
    o.order_date,
    o.status,
    c.customer_id,
    c.name,
```

```
    c.email,
    c.phone,
    p.warehouse_id,
    w.name;
```

5. `order_detail_summary`

- Provides complete order details including customer, product, quantity, and timestamps.
- Supports customer service and operational analytics.

```
CREATE OR REPLACE VIEW order_detail_summary AS
SELECT
    o.order_id,
    o.order_date,
    o.status AS order_status,

    c.customer_id,
    c.name AS customer_name,
    c.email AS customer_email,
    c.phone AS customer_phone,

    oi.order_item_id,

    p.product_id,
    p.name AS product_name,
    p.price AS product_price,
    poi.quantity AS quantity_ordered,
    poi.total_price AS total_price,

    w.warehouse_id,
    w.name AS warehouse_name,

    o.created_time AS order_created_time,
    o.updated_time AS order_updated_time
FROM
    orders o
    JOIN customers c ON o.customer_id = c.customer_id
    JOIN order_items oi ON o.order_id = oi.order_id
    JOIN product_order_items poi ON oi.order_item_id = poi.order_item_id
    JOIN products p ON poi.product_id = p.product_id
    JOIN warehouses w ON p.warehouse_id = w.warehouse_id;
```

6. `product_summary`

- Lists all products along with category, supplier, and warehouse details.
- Useful for both staff and admin product listings and searches.

```
CREATE OR REPLACE VIEW product_summary AS
SELECT
    p.product_id,
    p.name,
    p.description,
    p.price,
    p.image_url,
    p.quantity,
    c.category_id AS category_id,
    c.name AS category_name,
    s.supplier_id AS supplier_id,
```

```
    s.name AS supplier_name,
    w.warehouse_id AS warehouse_id,
    w.name AS warehouse_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
JOIN suppliers s ON p.supplier_id = s.supplier_id
JOIN warehouses w ON p.warehouse_id = w.warehouse_id
ORDER BY p.updated_time DESC, product_id ASC;
```

7. `user_activity`

- Tracks login logs by user with details on IP address, user agent, and warehouse.
- Helps with user monitoring and security audits.

```
CREATE OR REPLACE VIEW user_activity AS
SELECT
    u.user_id,
    u.username,
    u.email,
    u.role_name,
    w.name AS warehouse_name,
    ll.login_time,
    ll.ip_address,
    ll.user_agent
FROM users u
LEFT JOIN login_logs ll ON u.user_id = ll.user_id
LEFT JOIN warehouses w ON u.warehouse_id = w.warehouse_id;
```

8. `low_stock_alert`

- Flags products with quantity below threshold (10).
- Used to trigger restocking actions.

```
CREATE OR REPLACE VIEW low_stock_alert AS
SELECT
    p.product_id,
    p.name AS product_name,
    p.quantity,
    w.name AS warehouse_name,
    s.name AS supplier_name
FROM products p
JOIN warehouses w ON p.warehouse_id = w.warehouse_id
JOIN suppliers s ON p.supplier_id = s.supplier_id
WHERE p.quantity < 10;
```

2.3 Stored Procedures

For our project, we only use 2 procedure only

2.3.1 CreateOrder

Purpose: This procedure creates a new order for a customer, inserts the associated order item, and links multiple products with their quantities and total prices to the order item.

```
DELIMITER //

CREATE PROCEDURE CreateOrder(
```

```
    IN p_customer_id INT,
    IN p_product_ids TEXT,
    IN p_quantities TEXT,
    IN p_prices TEXT
)
BEGIN
    DECLARE order_id INT;
    DECLARE order_item_id INT;

    INSERT INTO orders (customer_id) VALUES (p_customer_id);
    SET order_id = LAST_INSERT_ID();

    INSERT INTO order_items (order_id) VALUES (order_id);
    SET order_item_id = LAST_INSERT_ID();

    WHILE LOCATE(',', p_product_ids) > 0 DO
        INSERT INTO product_order_items (product_id, order_item_id, quantity, total_price)
        VALUES (
            CAST(SUBSTRING_INDEX(p_product_ids, ',', 1) AS UNSIGNED),
            order_item_id,
            CAST(SUBSTRING_INDEX(p_quantities, ',', 1) AS UNSIGNED),
            CAST(SUBSTRING_INDEX(p_prices, ',', 1) AS DECIMAL(10, 2))
        );

        SET p_product_ids = SUBSTRING(p_product_ids FROM LOCATE(',', p_product_ids) + 1);
        SET p_quantities = SUBSTRING(p_quantities FROM LOCATE(',', p_quantities) + 1);
        SET p_prices = SUBSTRING(p_prices FROM LOCATE(',', p_prices) + 1);
    END WHILE;

    INSERT INTO product_order_items (product_id, order_item_id, quantity, total_price)
    VALUES (
        CAST(p_product_ids AS UNSIGNED),
        order_item_id,
        CAST(p_quantities AS UNSIGNED),
        CAST(p_prices AS DECIMAL(10, 2))
    );
END //

DELIMITER ;
```

2.3.2 UpdateProductQuantity

Purpose: This procedure updates the quantity of a product in stock after an order has been placed or modified.

```
DELIMITER //

CREATE PROCEDURE UpdateProductQuantity(
    IN p_product_id INT,
    IN p_quantity_change INT
)
BEGIN
    UPDATE products
    SET quantity = quantity - p_quantity_change
    WHERE product_id = p_product_id;
END //

DELIMITER ;
```

2.4 Triggers

To ensure product quantity reflects real-time sales, we implement a trigger to auto-decrement inventory when an order is placed.

```
DELIMITER $$

CREATE TRIGGER trg_after_insert_product_order_items
AFTER INSERT ON product_order_items
FOR EACH ROW
BEGIN
    UPDATE products
    SET quantity = quantity - NEW.quantity,
        updated_time = CURRENT_TIMESTAMP
    WHERE product_id = NEW.product_id;
END$$

DELIMITER ;
```

- **Purpose:** Automatically updates product quantity after an item is ordered.
- **Scope:** Applies to all inserts into `product_order_items`.

3 Performance tuning

3.1 System Architecture

This system is designed using a microservices-based architecture with containerized deployment. It integrates a relational database, in-memory data store, object storage, and web frontend to deliver a complete inventory and order management solution.

Backend Services:

- Implemented with **FastAPI**, exposing RESTful APIs for core modules (Product, Order, User, etc.).
- Each service is deployed as a separate container and communicates via internal network.

Databases:

- **MySQL**: Stores normalized relational data with support for constraints, stored procedures, triggers, and views.
- **Redis**: Used for caching and pub/sub messaging to support real-time synchronization (e.g., between frontend dashboards).
- **MinIO**: S3-compatible object storage for hosting product images, invoices, and exported reports.

Frontend:

- Built with **Next.js**, a React framework for server-side rendering and API integration.
- Interacts with backend services through API endpoints.

Authentication and Validation:

- **JWT (JSON Web Tokens)** is used for stateless user authentication.
- **Pydantic** ensures request/response schema validation in FastAPI.

Deployment and DevOps:

- **Docker** and **Docker Compose** manage containerized services and networks.
- **Nginx Proxy Manager** acts as a reverse proxy to route public traffic to service endpoints.
- **Portainer** is used for visual container management.
- **GitHub Actions** handles CI/CD for building, testing, and deploying services automatically.

Setup and Usage:

1. Clone the project repository:

```
git clone git@github.com:truongng201/Database-IMS-Project.git
cd Database-IMS-Project
```

2. Configure environment variables using ‘.env‘ (refer to ‘.template.env‘ for structure).
3. Build and launch backend and frontend services:

```
make services-up SERVICE='product order'  # Example: start product and order services
make client-dev                            # Launch frontend in development mode
```

4. Access the application locally:
   - Frontend: http://localhost:3000
   - API Services: http://localhost:8080/v1/{service_name}
   - MinIO UI: http://localhost:9001

5. Access the application in production (DuckDNS setup):
    - Frontend: https://vinuni-database-ims.duckdns.org
    - API: https://api.vinuni-database-ims.duckdns.org/v1/{service_name}
    - Documentation: https://api.vinuni-database-ims.duckdns.org/v1/user/docs
    - MinIO: https://minio.vinuni-database-ims.duckdns.org
    - Portainer: https://portainer.vinuni-database-ims.duckdns.org
    - Proxy Manager: https://proxy-manager.vinuni-database-ims.duckdns.org

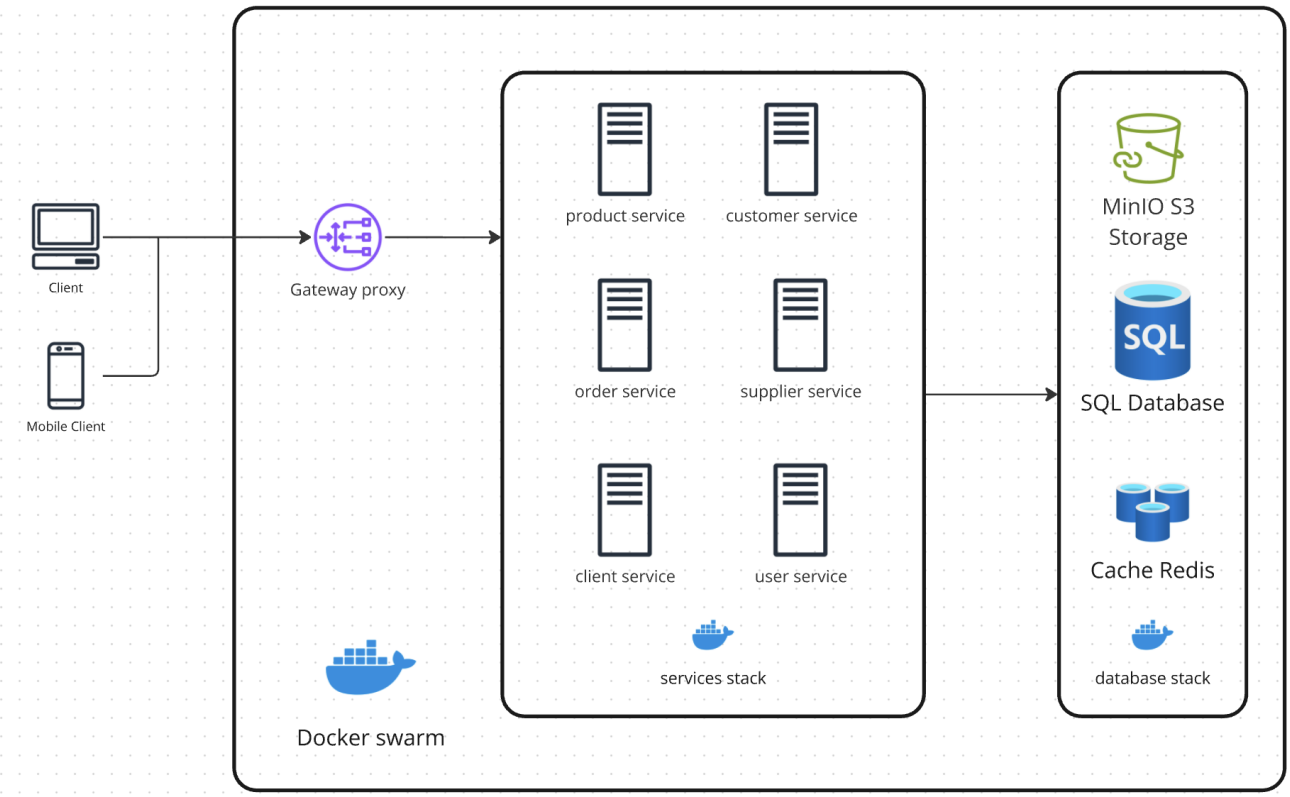


Figure 2: Overall system architecture

3.2 Indexing

3.2.1 Use of Indexing

Indexing involves creating special lookup structures (indexes) on one or more columns of a table. These indexes significantly improve data retrieval speed by allowing the database engine to locate rows more efficiently, rather than performing a full table scan. Indexes are particularly beneficial for columns that are frequently used in `WHERE` clauses, `JOIN` conditions, or `ORDER BY` clauses.

3.2.2 Query Optimization Evidence (Before vs. After)

One sample case: Optimizing customer Lookup by Email (`customers.email`)

- **Objective:** To improve the speed and efficiency of querying customer information based on their email address, a frequent operation for customer service, order processing, or marketing purposes.
- **Strategy:** An index named `idx_customers_email` was created on the `email` column of the `customers` table.
- **Analysis (Before):**

– The `EXPLAIN` output indicated a full table scan (`type: ALL`) due to the absence of a suitable index on the `email` column.

– A high `Handler_read_rnd_next` value (101) confirmed that rows were being read through physical scanning, which is inefficient especially for large datasets.

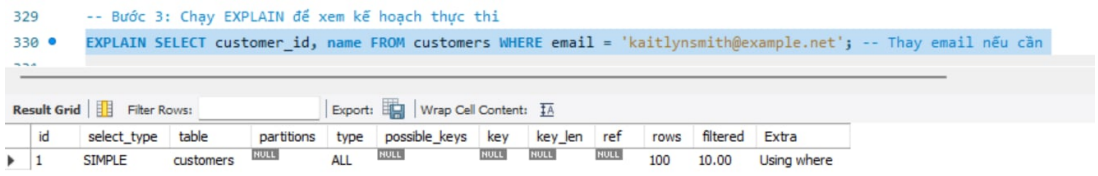


Figure 3: EXPLAIN output before indexing

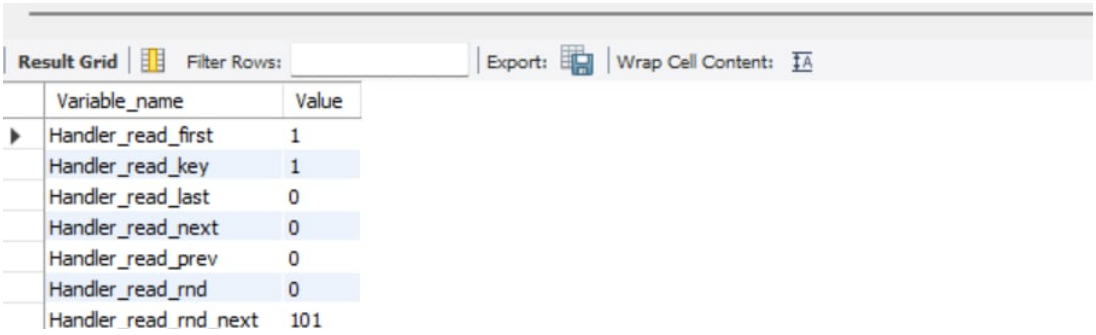


Figure 4: EXPLAIN output before indexing

- **Analysis (After):**
  – A significant improvement was observed: `EXPLAIN` showed a switch to an efficient index lookup (`type: ref`) using `idx_customers_email`.
  – Estimated rows scanned dropped from 100 to just 1.
  – `Handler_read_rnd_next` dropped to 0, eliminating the full table scan.
  – Increase in `Handler_read_key` (to 144) and `Handler_read_next` (to 117) was expected, reflecting indexed navigation.

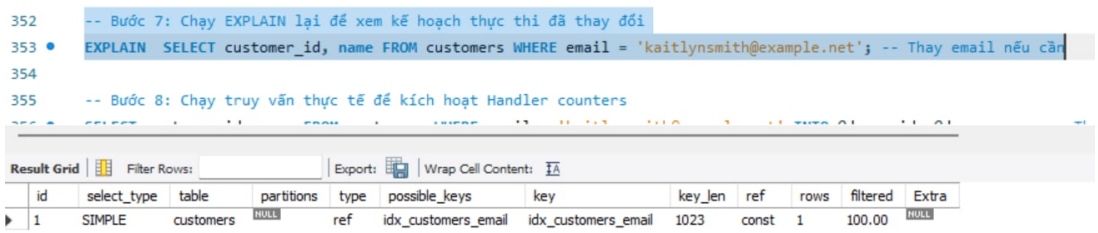


Figure 5: EXPLAIN output after indexing

```
359 •  SELECT '-- Test 1: Handler Counters SAU khi có index --' AS TC1_HANDLER_COUNTERS_AFTER;
360 •  SHOW SESSION STATUS LIKE 'Handler_read%';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Variable_name	Value
Handler_read_first	0
Handler_read_key	144
Handler_read_last	0
Handler_read_next	117
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_next	0

Figure 6: EXPLAIN output after indexing

- **Conclusion:** The optimization successfully shifted the query execution from an inefficient full table scan to a highly targeted index-based access pattern, significantly improving performance.

4 Security Configuration

This section outlines the core security mechanisms implemented in our Inventory Management System (IMS), focusing on access control, password storage, and SQL injection prevention.

4.1 Access Control & Authorization

The IMS system allows two types of users: **Admin** and **Staff**. Each user logs in using their `email` and `password` credentials. Upon successful login, role-based access control is enforced across the system to restrict sensitive operations and data access.

Staff Access Activation. Staff members must first be activated by an admin. Additionally, they are assigned a specific warehouse. They only have access to resources and information related to the assigned warehouse. This ensures compartmentalization of data and limits privilege scope.

Token-Based Access. To enforce secure access control:

- We use **JWT (JSON Web Tokens)** as access tokens.
- A **64-character random string** is used as the refresh token.

Access Token contains the following claims:

- `user_id`
- `email`
- `role` (admin or staff)
- `warehouse_id` (only if assigned)

This token must be presented when accessing any protected route. It is verified using a custom middleware function called `login_required`, defined at:

https://github.com/truongng201/Database-IMS-Project/blob/main/services/shared/template/shared_utils/middleware.py

This middleware performs the following:

- Validates the access token's signature and expiration.
- Extracts role and warehouse information.
- Restricts access based on the user's permissions.
- Checks if the token is blacklisted by consulting the cache system.

Logout Mechanism. A secure logout mechanism is implemented with the following steps:

1. When a user logs out, their refresh token is deleted from the `login_log` table in the database.
2. The corresponding access token is added to a **blacklist stored in the cache system**.
3. This blacklisted access token remains in the cache until it expires (access tokens are valid for 1 hour).
4. The middleware checks if the presented access token is in the blacklist cache. If it exists in the cache, the request is denied, even if the token is otherwise valid.

This logout flow ensures that a user session is securely terminated and prevents reuse of access tokens after logout.
All user authentication and authorization logic (register, login, token handling, logout) is encapsulated in the User Service module:

https://github.com/truongng201/Database-IMS-Project/tree/main/services/user

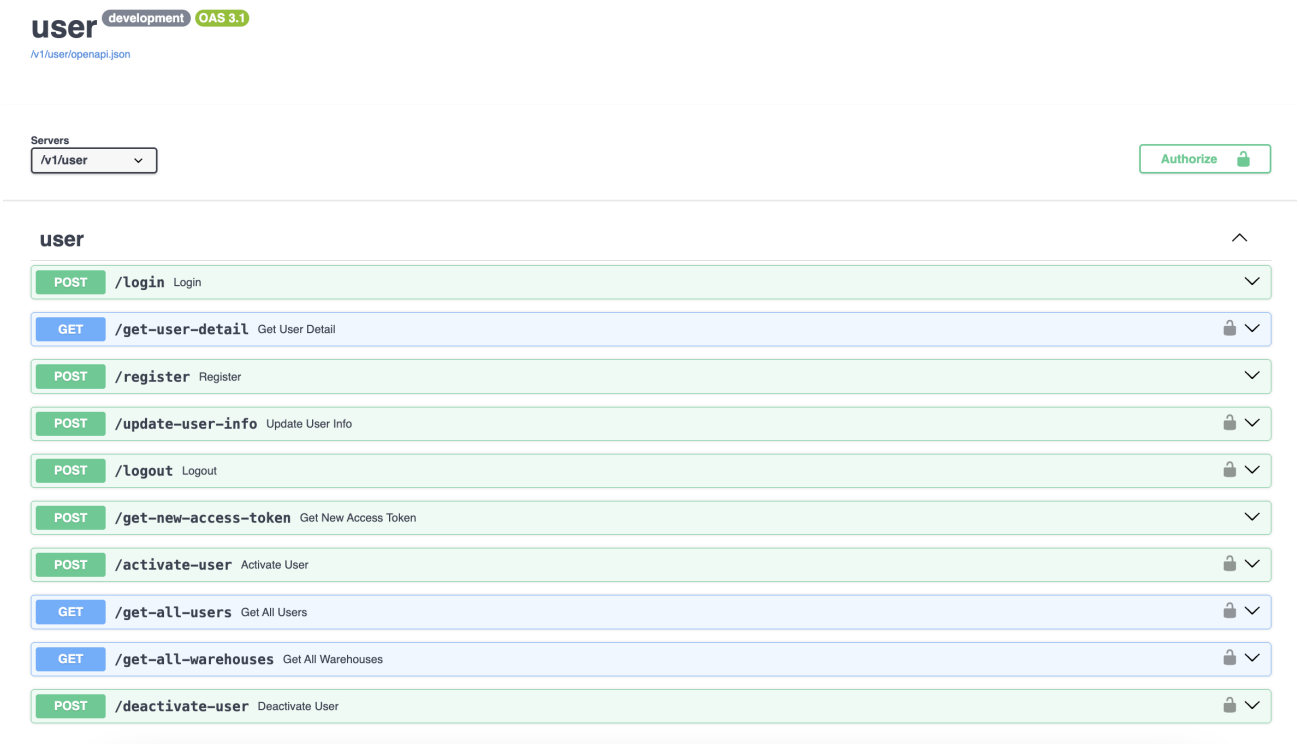


Figure 7: Swagger API documentation for User Service endpoints

4.2 Password Storage

To ensure secure storage of user credentials, passwords are **hashed using bcrypt** before being saved into the database.

Why bcrypt?

- **Salting**: Bcrypt automatically adds a unique salt to each password, which protects against rainbow table attacks.
- **Work Factor**: It includes a configurable cost factor, which allows increasing the computational time required for hashing as hardware improves.
- **Resilience**: It is specifically designed to be slow and thus resistant to brute-force attacks.

Why not MySQL hash functions? While MySQL offers built-in hashing functions like `MD5()` or `SHA1()`, these are:

- **Cryptographically Weak**: MD5 and SHA1 are outdated and vulnerable to collision and pre-image attacks.
- **No Salt Support**: These functions do not automatically salt the password, making them highly insecure.
- **Not Adjustable**: They lack tunable work factors to adapt to future hardware.

Hence, password hashing is handled at the application level using bcrypt before storage in the MySQL database.

4.3 Preventing SQL Injection

SQL injection is one of the most common and dangerous security vulnerabilities in database-driven applications. To mitigate this risk, the IMS uses **prepared statements with parameterized queries** throughout all database interactions.

What are prepared statements? Prepared statements separate the SQL code from the data input by the user. Placeholders are used in the SQL query, and actual values are passed separately. This eliminates the risk of malicious SQL code being injected into queries.

Advantages:

- **Protection**: Prevents malicious user input from altering the SQL command structure.
- **Reusability**: Compiles the SQL statement once and reuses it with different inputs.
- **Performance**: Boosts performance on frequently executed queries.

```python
def get_all_product_by_user(self, user_id, params=(100, 0), search=None):
    # First get the user's warehouse_id
    user_query = """
    SELECT warehouse_id FROM users WHERE user_id = %s
    """
    user_result = self.db.execute_query(user_query, (user_id,))
    if not user_result:
        return []

    warehouse_id = user_result[0][0]

    where_conditions = ["warehouse_id = %s"]
    query_params = [warehouse_id]

    # Build search conditions - search across name, category, and supplier
    if search:
        where_conditions.append("(name LIKE %s OR category_name LIKE %s OR supplier_name LIKE %s)")
        search_term = f"%{search}%"
        query_params.extend([search_term, search_term, search_term])

    # Build query
    query = "SELECT * FROM product_summary WHERE " + " AND ".join(where_conditions)
    query += " LIMIT %s OFFSET %s"
```

Figure 8: Example of using prepared statements in Python (MySQL connector)

Example: This approach is used across all services in the project to maintain consistency and safeguard against injection vulnerabilities.

5 End-to-end Testing & Web Integration

Note: All core CRUD functionalities have been fully implemented. However, due to the project's large scope and limited time constraints, a few features are currently either completed on the API side or on the client side, but not yet fully integrated between the two. Integration for these remaining parts is planned for future updates.

To validate the core functionalities of the Inventory Management System (IMS), we conducted comprehensive end-to-end testing that simulates realistic workflows across different user roles and services. This section documents the testing process and outcomes.

5.1 Sample Data

For testing purposes, we generated a set of realistic sample data using a custom Python script available at:

https://github.com/truongng201/Database-IMS-Project/blob/main/scripts/generate_fake_data.py

The generated data includes:

- **NUM_SUPPLIERS = 100**
- **NUM_WAREHOUSES = 5**
- **NUM_PRODUCTS = 1000**
- **NUM_CUSTOMERS = 100**
- **NUM_ORDERS = 1000**
- **NUM_ORDER_ITEMS = 1000**

```
INSERT INTO categories (name, description) VALUES ('Electronics', 'Devices, gadgets, and accessories.');
INSERT INTO categories (name, description) VALUES ('Books', 'Printed and digital reading materials.');
INSERT INTO categories (name, description) VALUES ('Clothing', 'Men''s and women''s apparel.');
INSERT INTO categories (name, description) VALUES ('Toys', 'Children''s play items.');
INSERT INTO categories (name, description) VALUES ('Furniture', 'Home and office furnishings.');
INSERT INTO categories (name, description) VALUES ('Sports', 'Equipment and apparel for sports activities.');
INSERT INTO categories (name, description) VALUES ('Beauty', 'Cosmetics and personal care products.');
INSERT INTO categories (name, description) VALUES ('Automotive', 'Car parts and accessories.');
INSERT INTO categories (name, description) VALUES ('Food', 'Groceries and gourmet items.');
INSERT INTO categories (name, description) VALUES ('Health', 'Wellness and medical supplies.');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Rodriguez Inc', 'Rebecca Benson', 'msmith@nelson-patel.com', '(449)537-5791x8952');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Blair-Diaz', 'Michael Smith', 'areynolds@wheeler-gomez.com', '001-631-247-3351x90028');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Sherman, Davis and Choi', 'Andrew Brown', 'taylorvicki@wilson-ramirez.com', '297-333-7636');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Bean, Johnson and Robinson', 'John Collins', 'patrickmurphy@tucker.com', '625-572-8848x695');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Vasquez, Mcdonald and Lewis', 'Diana Brooks', 'mariejohnson@martin-thornton.com', '001-714-213-8030x223');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Shelton, Kim and Allen', 'Colin Lutz', 'rebeccabrown@hart.com', '(679)847-6845x08880');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Taylor LLC', 'Christina Allen', 'melissa22@henderson.com', '(416)771-5992');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Booth Group', 'Brittany Campos', 'shawnwright@parks.biz', '+1-232-590-3643x808');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Garcia Inc', 'Phillip Hoffman', 'smithdaniel@ward-johnson.com', '871-349-0205x49481');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Cox, Wolfe and Lyons', 'Jesse Mays', 'henryleon@cooper-pierce.org', '001-965-315-9500x6670');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Odom, Wallace and Bryan', 'Zachary Vargas', 'bpena@patton.com', '644-783-1306');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Wilson, Johnson and Watson', 'Charles Kelly', 'ewilliams@williams.com', '208-879-9764');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Casey-Mendoza', 'Walter Hanson', 'ashleystevens@webb.com', '001-847-215-7724x2246');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Diaz-Williams', 'Shannon Gonzalez', 'yreed@snyder-richard.net', '372-952-9691x5998');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Ramsey Ltd', 'Mark Fletcher', 'stephanie06@moore.com', '785-613-0843');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Flores-Bush', 'Matthew Hernandez', 'mackenzie67@williams.com', '(437)818-3687x12945');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Miller, Coffey and Harris', 'Paul Best', 'megan19@rodriguez-garcia.com', '3595250201');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('White, Solis and Martin', 'Rita Vasquez', 'xbaker@ray-leach.com', '286-492-9585');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Johns, Anderson and Collins', 'Jose Bradford', 'agutierrez@lane.info', '(274)906-9061x732');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Hammond-Rivera', 'Cynthia Chavez', 'gregorydurham@luna.com', '001-471-265-1914');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Sloan-Hill', 'Wendy Collins', 'tina02@davis.org', '317-552-7231x607');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Taylor Inc', 'William Jensen', 'nielsenkathleen@daugherty.biz', '3017984923');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Hinton and Sons', 'Jacqueline Andrews', 'zpitts@rodriguez.com', '001-832-698-5167x674');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Mitchell Ltd', 'Patricia Harrison', 'arodgers@cooper-barnett.org', '+1-375-287-4240x306');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Henderson-Hamilton', 'John Wolfe', 'elarsen@lynch.com', '901.476.7218x4758');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Bailey, Costa and Smith', 'Jeremy Chavez', 'nedwards@martin.net', '+1-270-697-4213x86156');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Malone-Tate', 'Richard Underwood', 'jclark@henderson.com', '(626)271-1793x912');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Patton, Greer and Gomez', 'Patrick Burnett', 'vwerner@krueger-king.net', '+1-462-878-2429x785');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Butler, Thomas and Mason', 'Melissa Johnson', 'rvillegas@lucas.biz', '(364)513-7929');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Ball-Collins', 'Raymond Washington', 'orhodes@villa.com', '764.691.5699x74072');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Lee and Sons', 'Donna Campbell', 'jamescole@powell.com', '(947)941-8942x7624');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Hernandez Inc', 'John Smith', 'morrislaura@madden.com', '772.929.4305');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Maxwell Inc', 'John Macias', 'christinashah@martin.org', '(368)683-8381x385');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Patterson Inc', 'Jennifer Turner', 'maureen49@montoya.com', '(737)268-2976x978');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Kim, Gibson and Padilla', 'Kathleen Welch', 'kingcaitlyn@murray.biz', '+1-475-825-9502x810');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Lee-Cruz', 'Gerald Reed', 'hmckinney@schmidt.com', '+1-980-592-2012x107');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Edwards-Thomas', 'Janet Roberts', 'laurenbell@harris.com', '+1-743-394-2835x0442');
INSERT INTO suppliers (name, contact_name, contact_email, phone) VALUES ('Owens-Wilkerson', 'Mario Daniels', 'katie51@boyer.com', '+1-739-421-2998');
```

Figure 9: Sample data view generated for test environment

5.2 User and Auth

5.2.1 Register

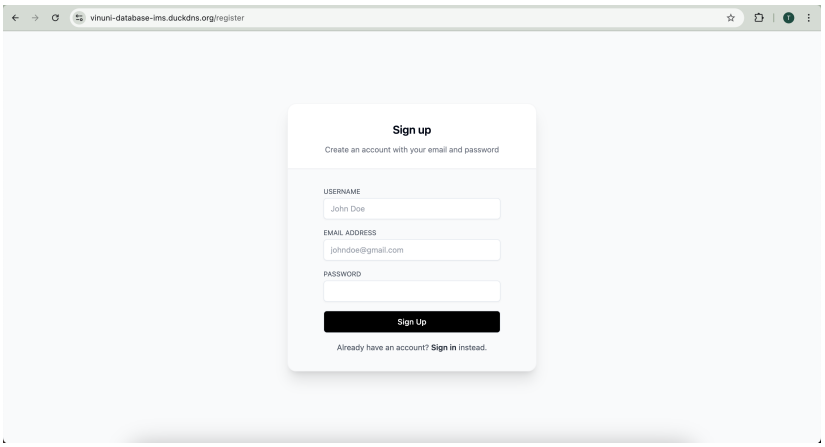


Figure 10: User registration page

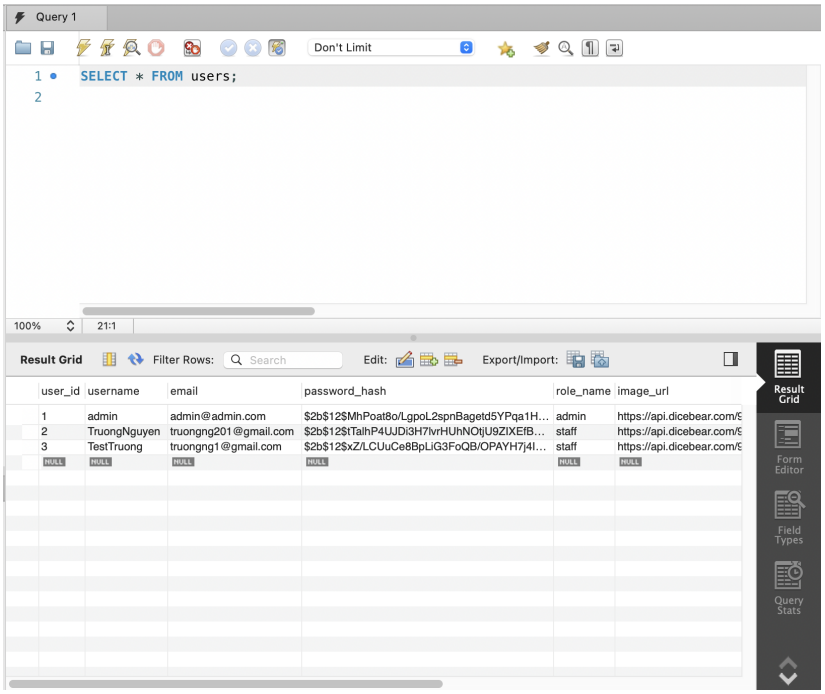


Figure 11: MySQL Workbench view showing hashed password with bcrypt

5.2.2 Login

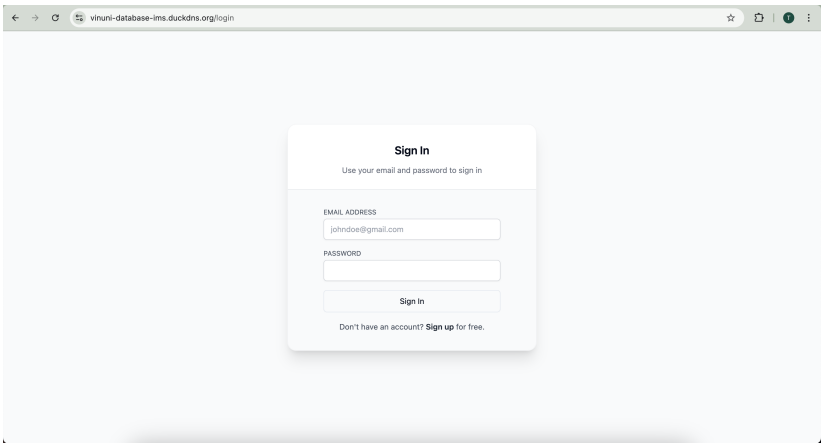


Figure 12: User login page

5.2.3 Login Account Not Activated

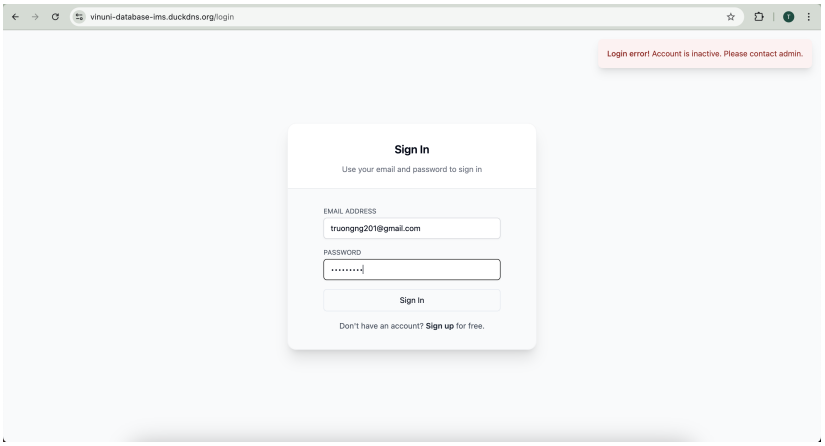


Figure 13: Error on login with inactive account

5.2.4 Login as Admin

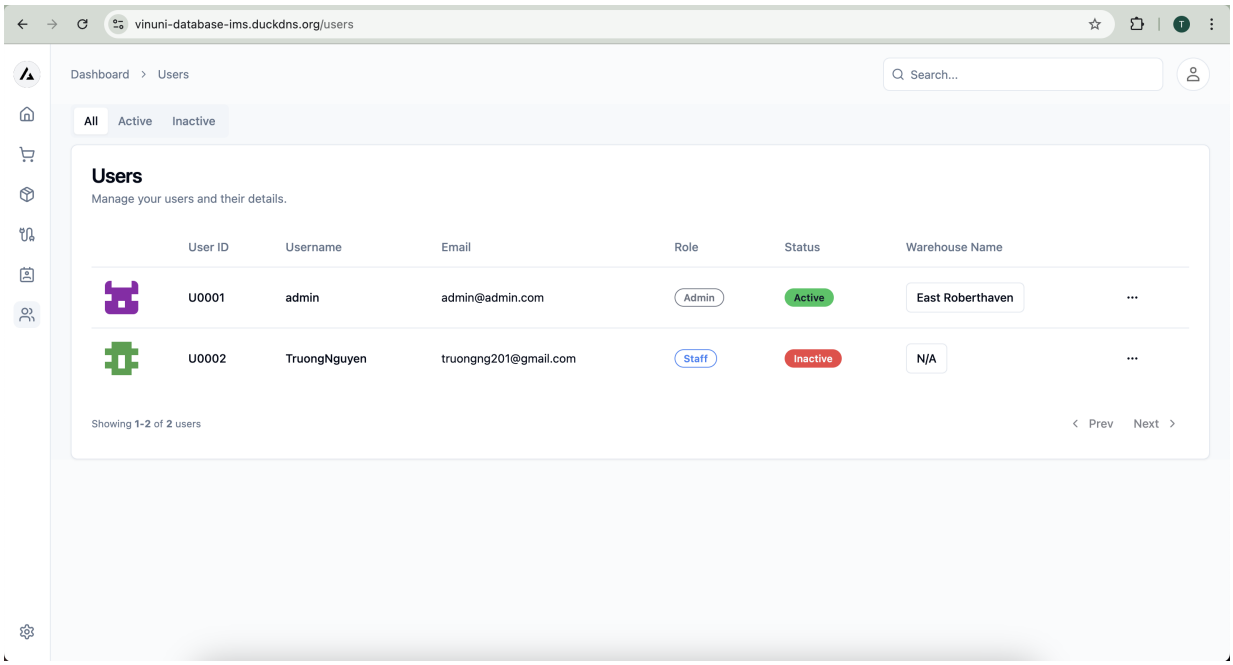


Figure 14: Admin view of all users

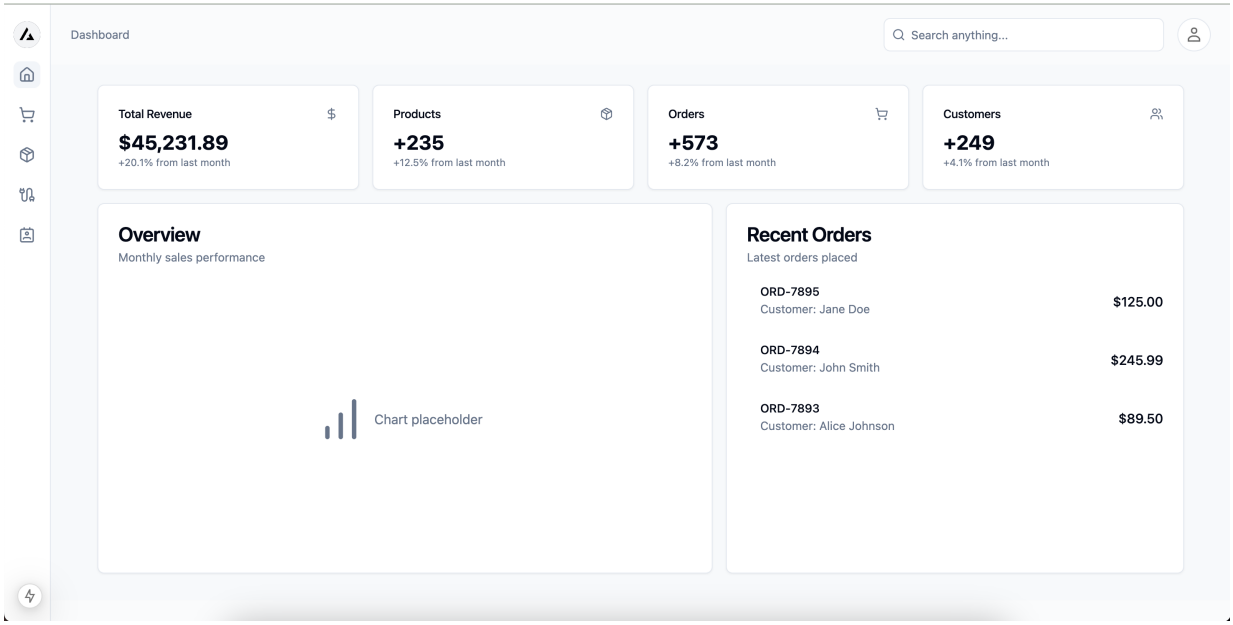


Figure 15: Dashboard view for Admin

5.2.5 Login as Staff

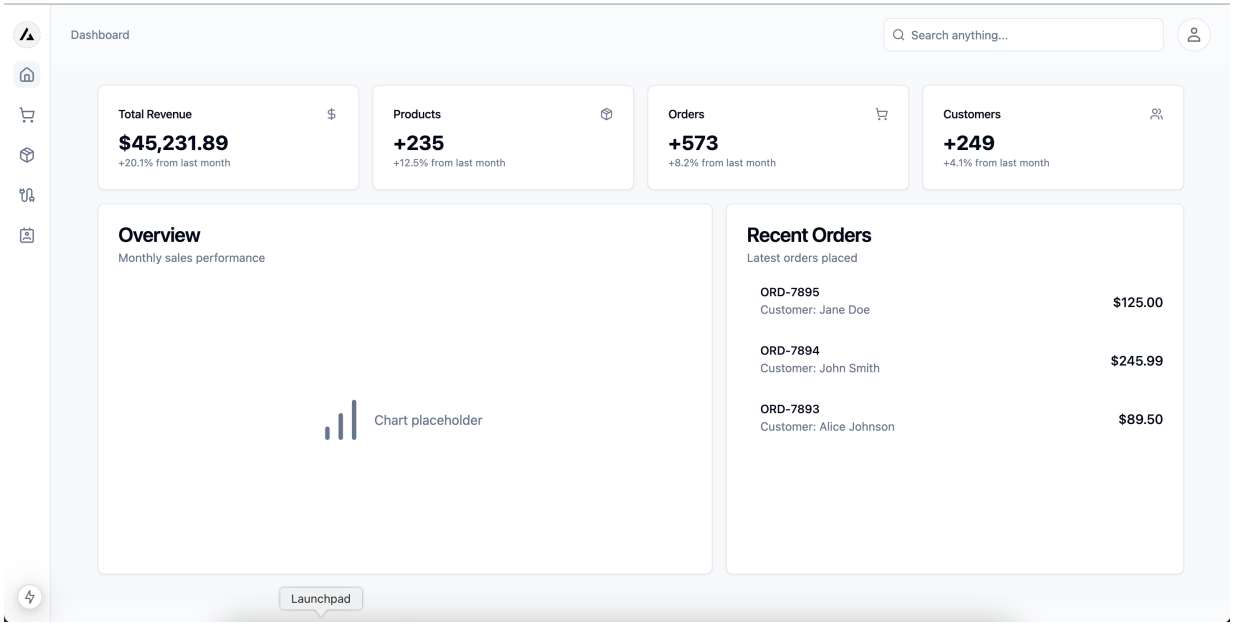


Figure 16: Dashboard view for Staff

Note: Both dashboards share the same layout, but content is filtered based on the assigned warehouse.

5.2.6 Logout

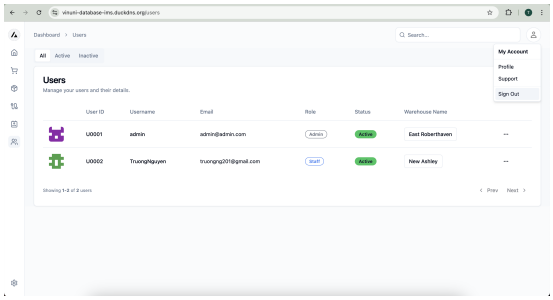


Figure 17: Logout button on dashboard

5.3 Product View

5.3.1 For Staff

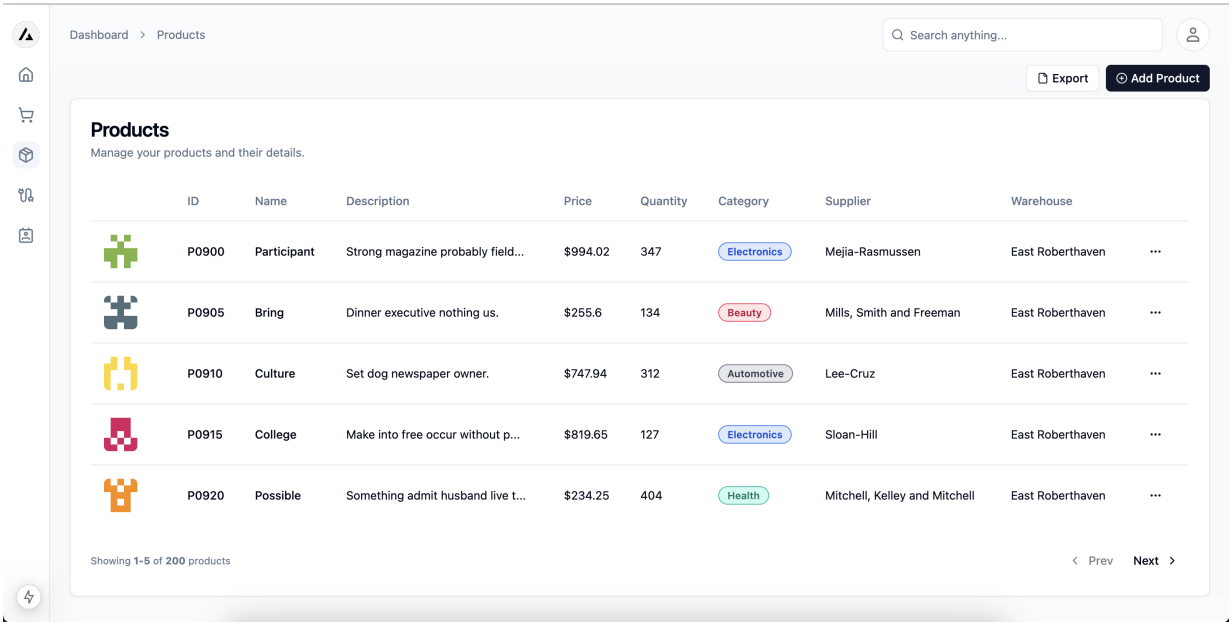


Figure 18: Product view limited to staff's assigned warehouse

5.3.2 For Admin

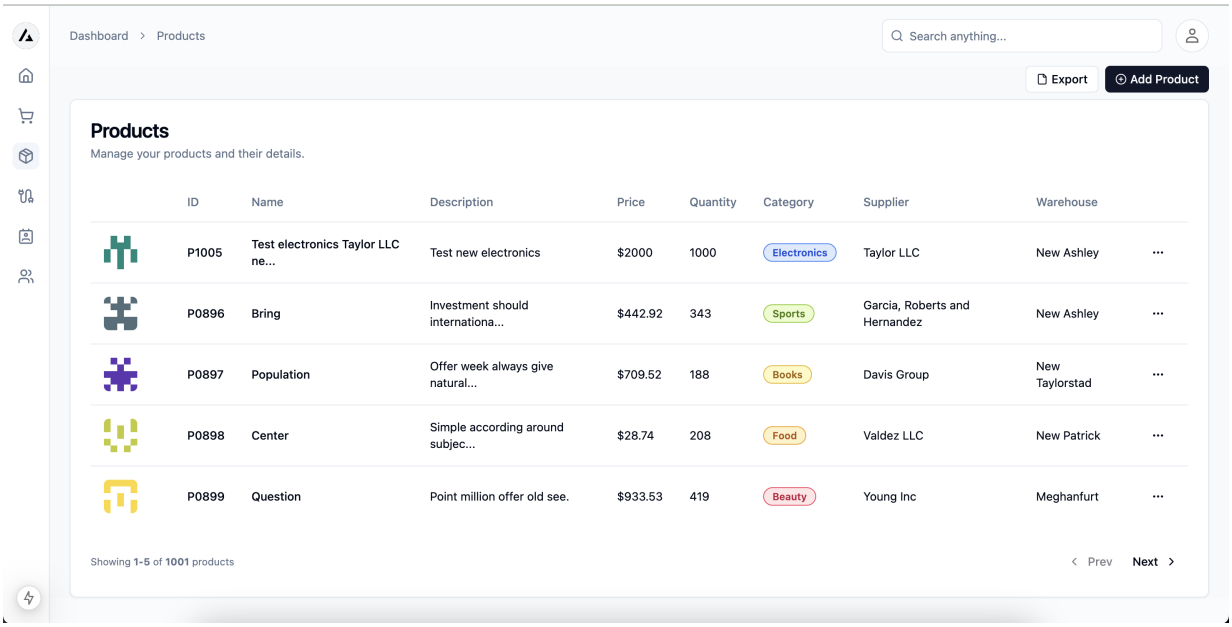


Figure 19: Product view for admin showing all products

5.3.3 Add Product

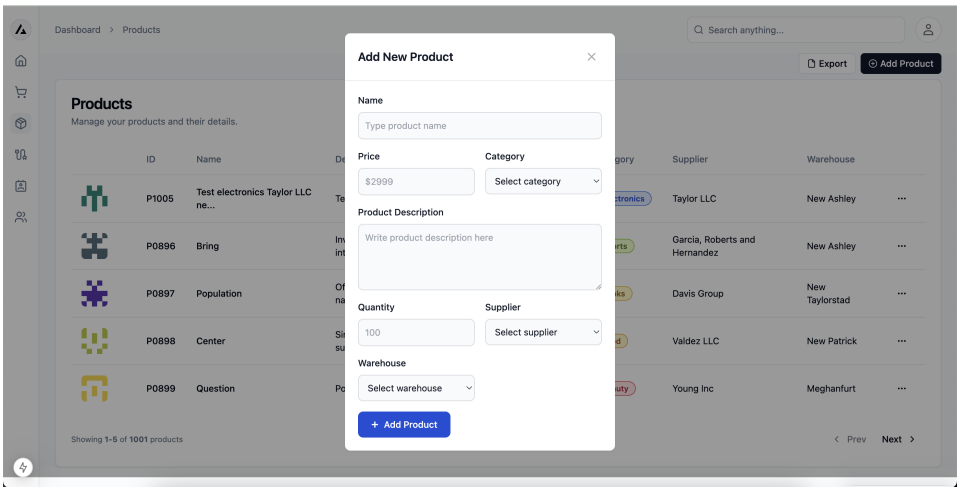


Figure 20: Add product form

5.3.4 Edit Product

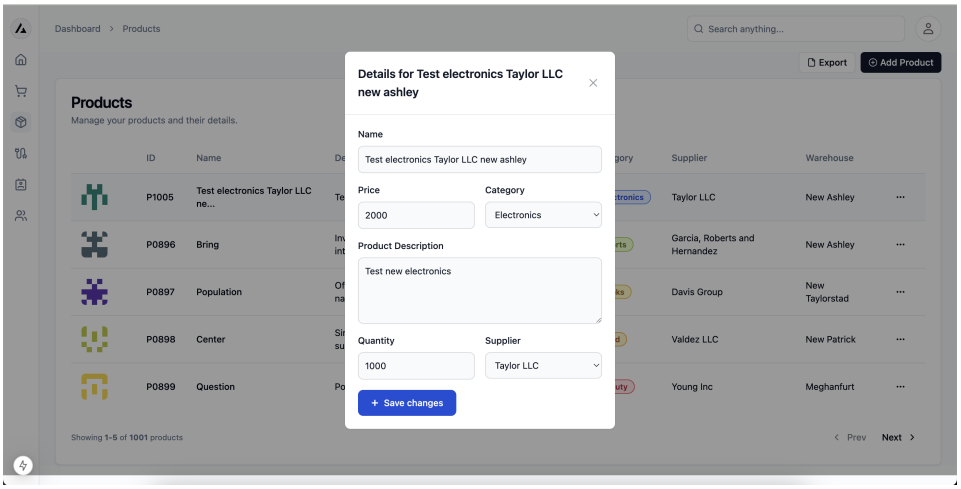


Figure 21: Edit product form

5.3.5 Super Search

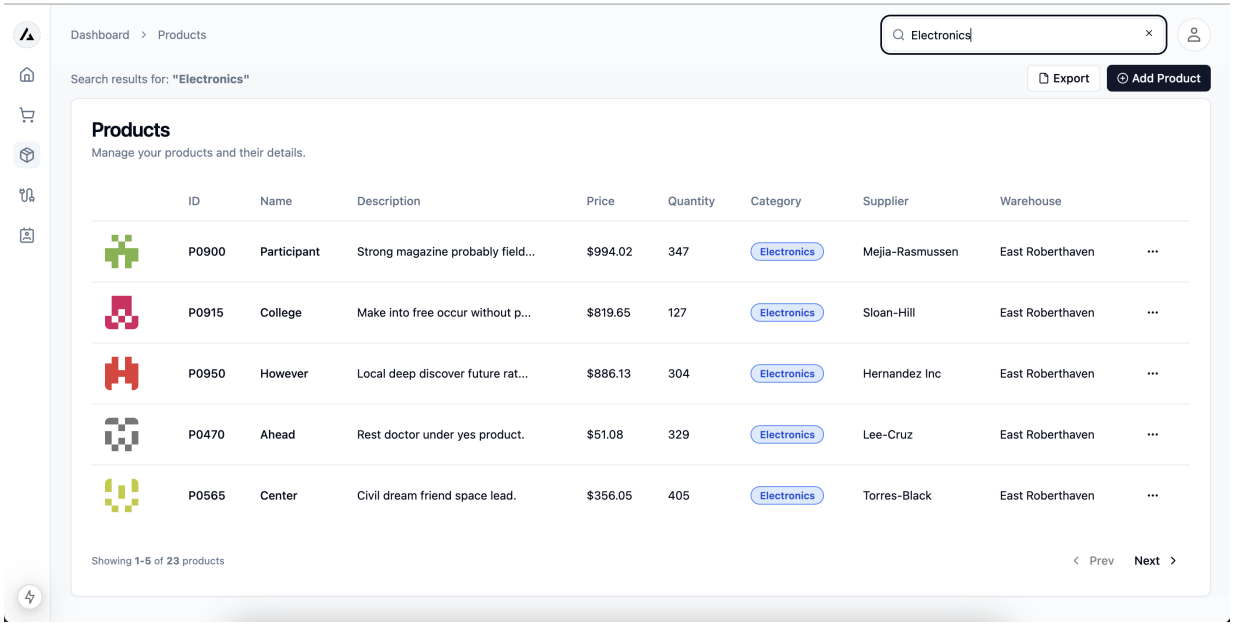


Figure 22: Super search - filtering by keyword 1

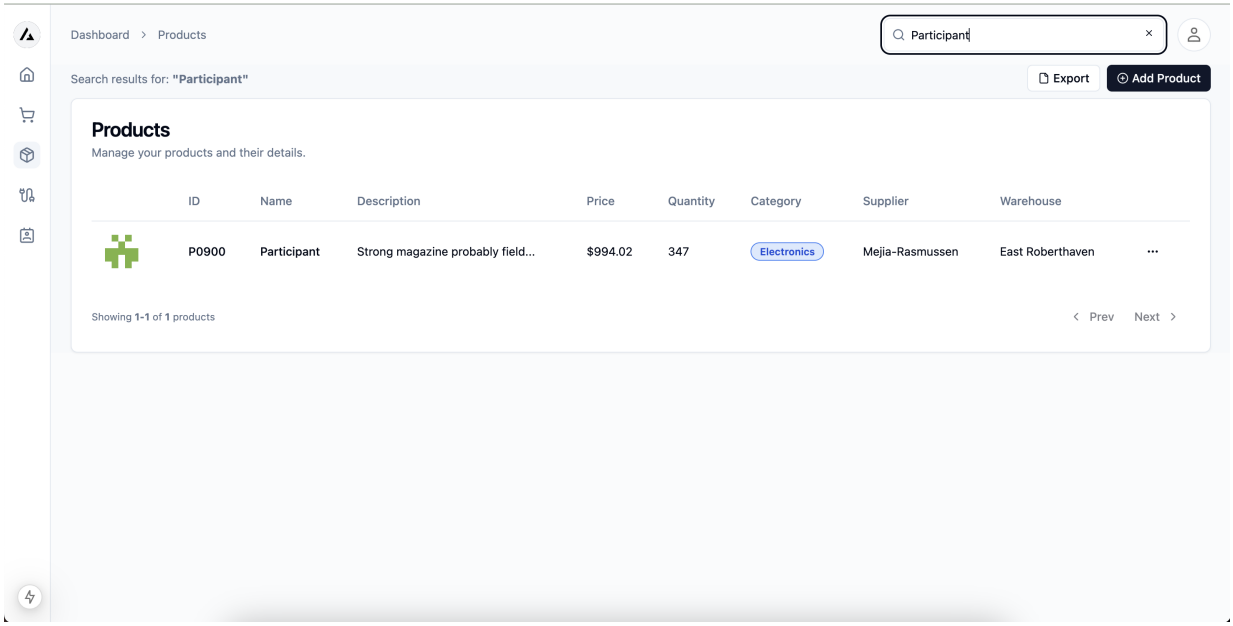


Figure 23: Super search - filtering by keyword 2

5.3.6 Product Endpoint

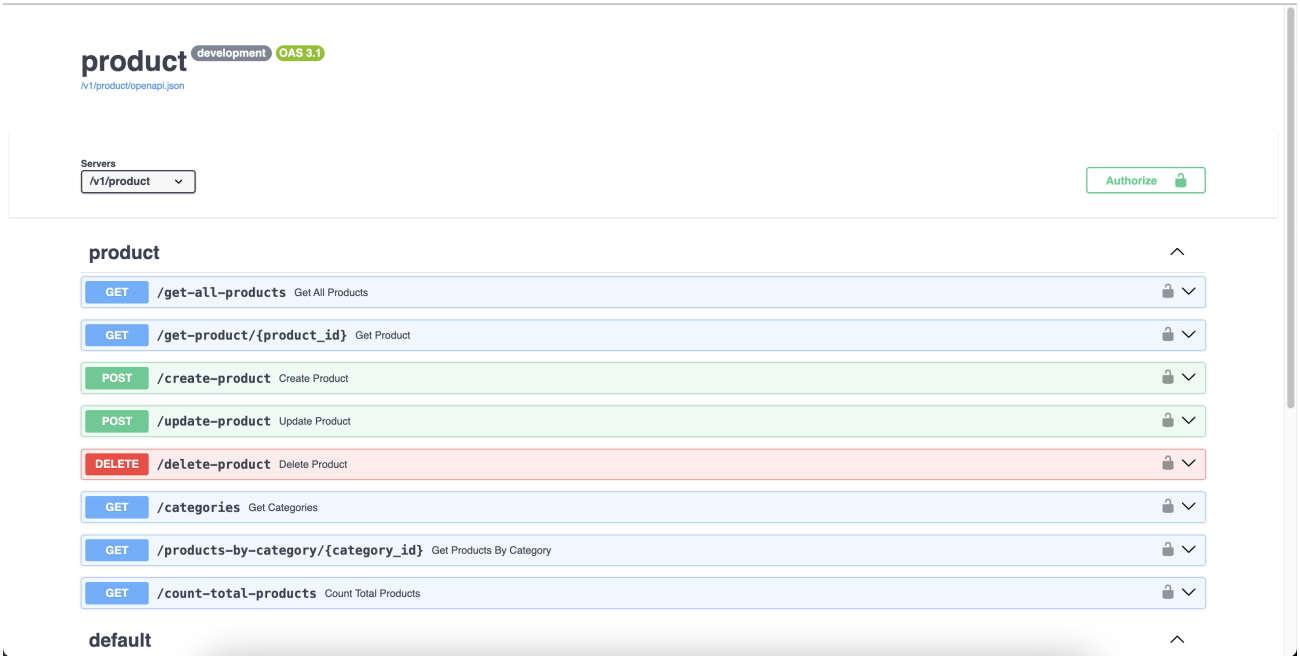


Figure 24: Swagger documentation for product API endpoints

5.4 Customer View

5.4.1 For Staff/Admin

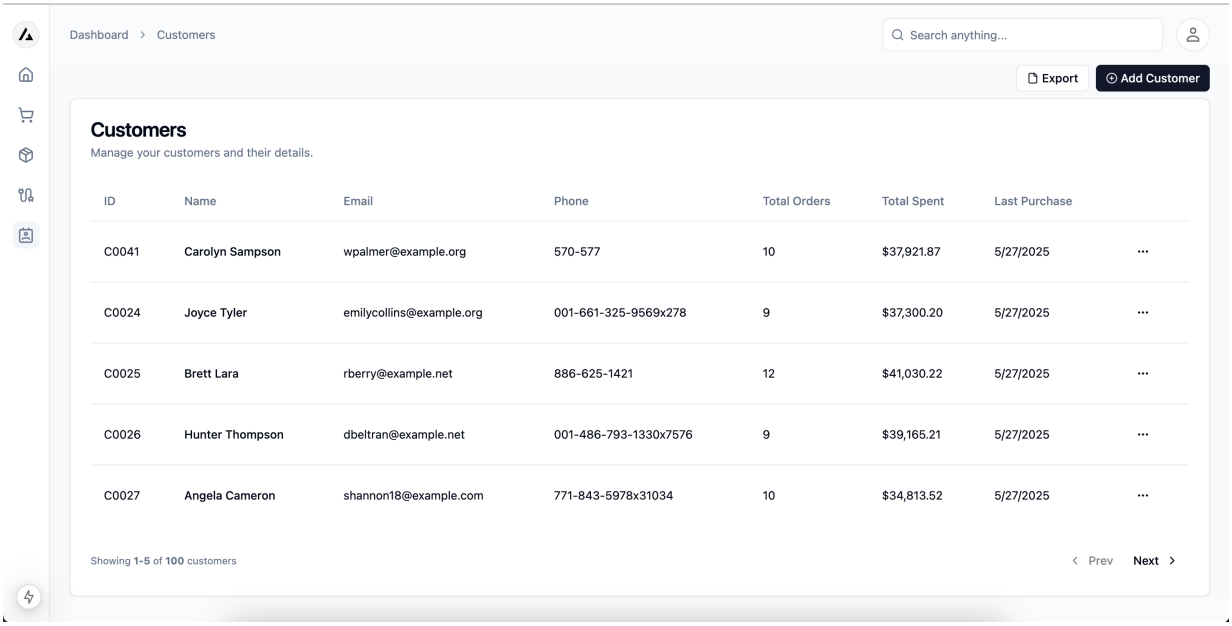


Figure 25: Customer list view

5.4.2 Add Customer

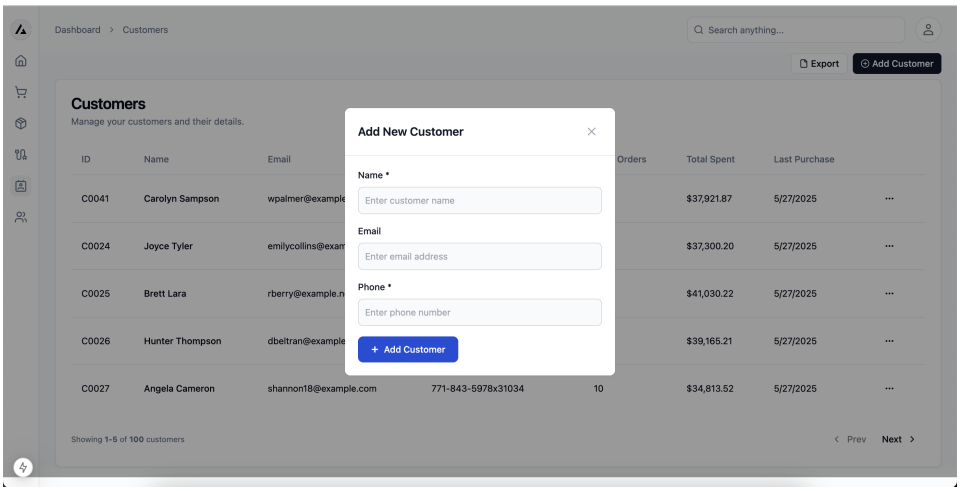


Figure 26: Add customer form

5.4.3 Edit Customer

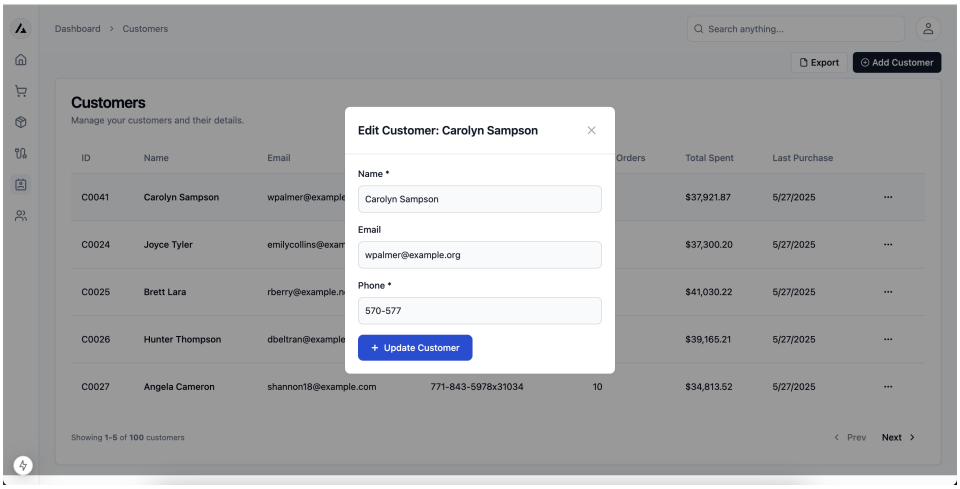


Figure 27: Edit customer form

5.4.4 Super Search

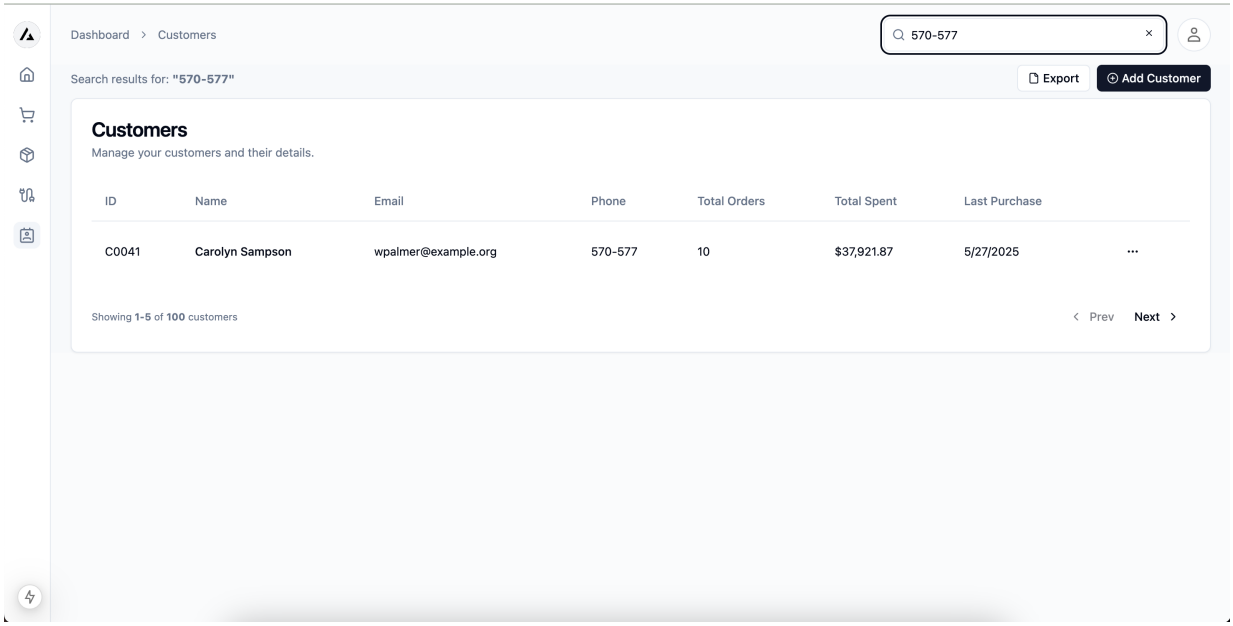


Figure 28: Super search - filtering by keyword 1

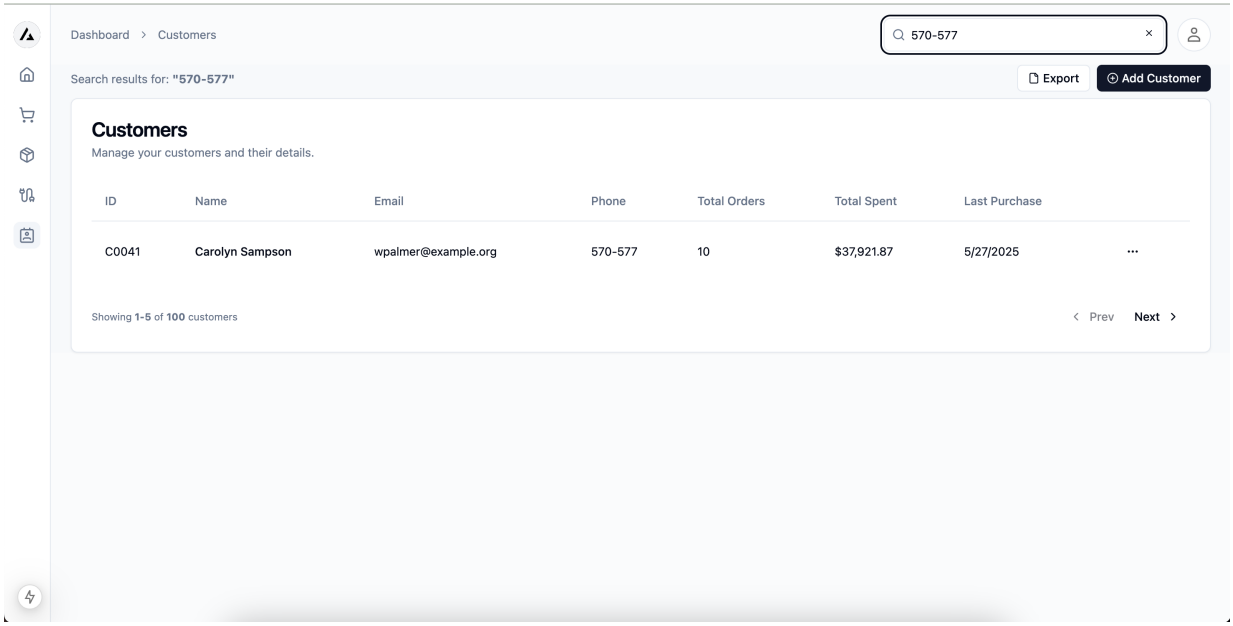


Figure 29: Super search - filtering by keyword 2

5.4.5 Customer Docs

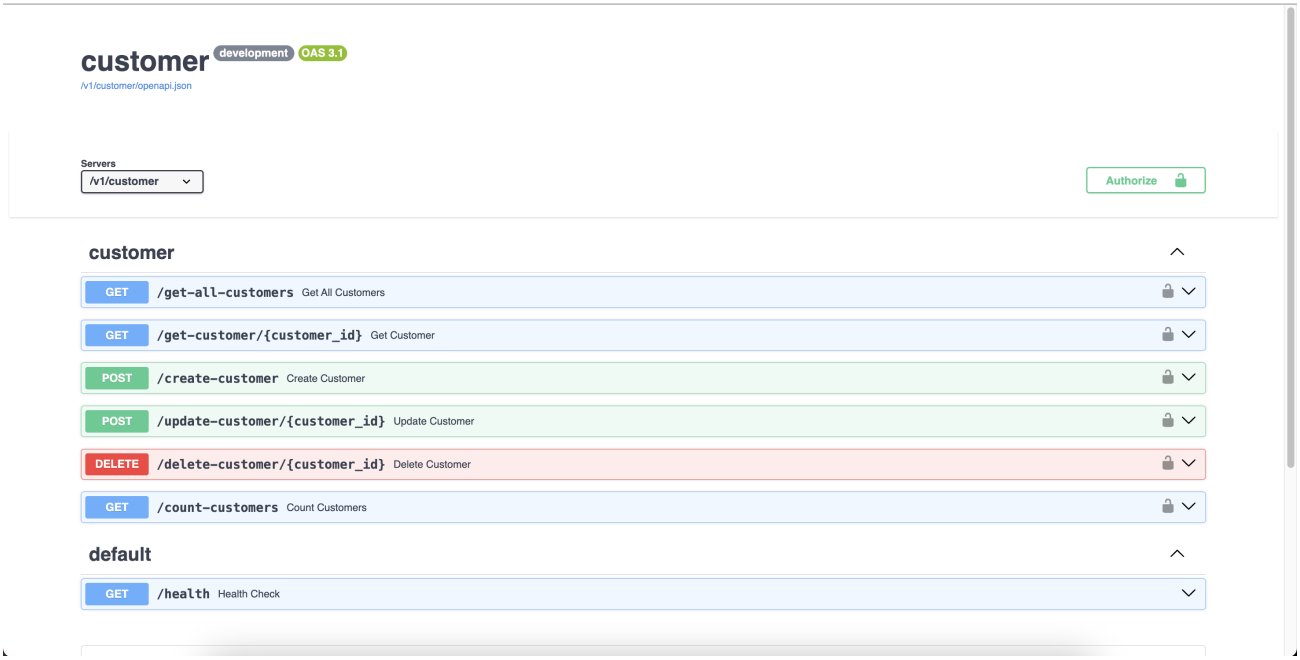


Figure 30: Customer API endpoint documentation

5.5 Supplier View

5.5.1 For Staff/Admin

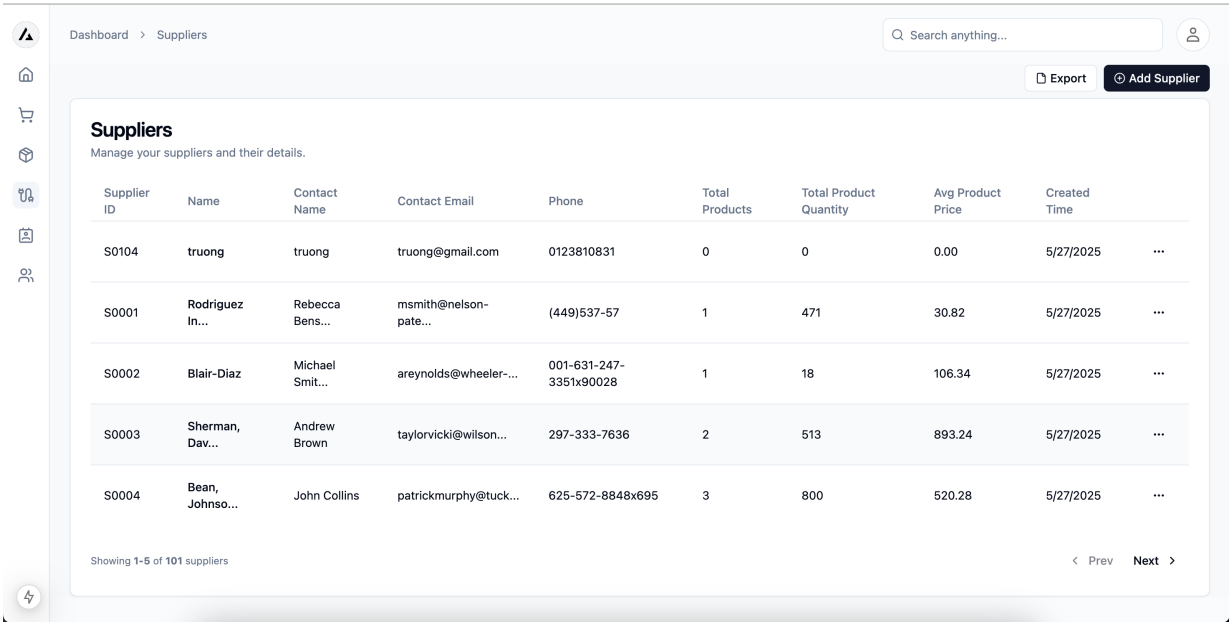


Figure 31: Supplier view for staff or admin

Information was shown differently for staff and admin

5.5.2 Add Supplier

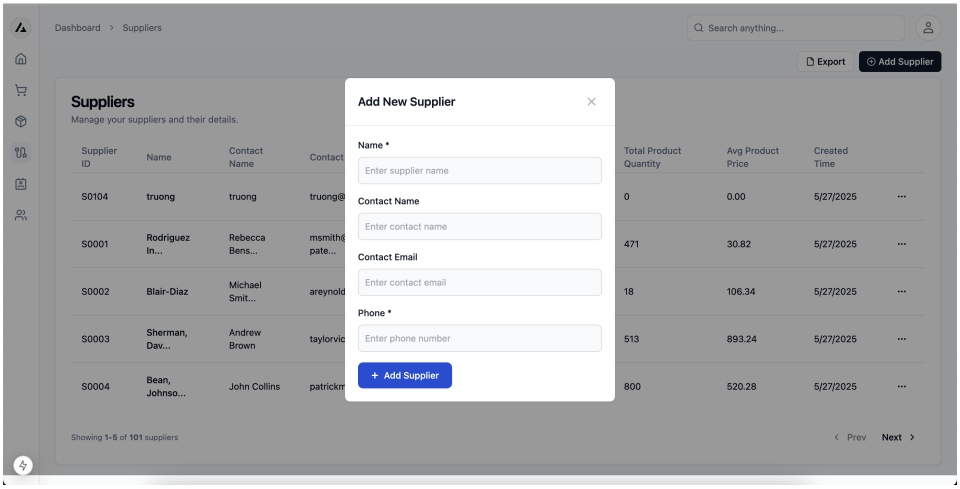


Figure 32: Add supplier form

5.5.3 Edit Supplier

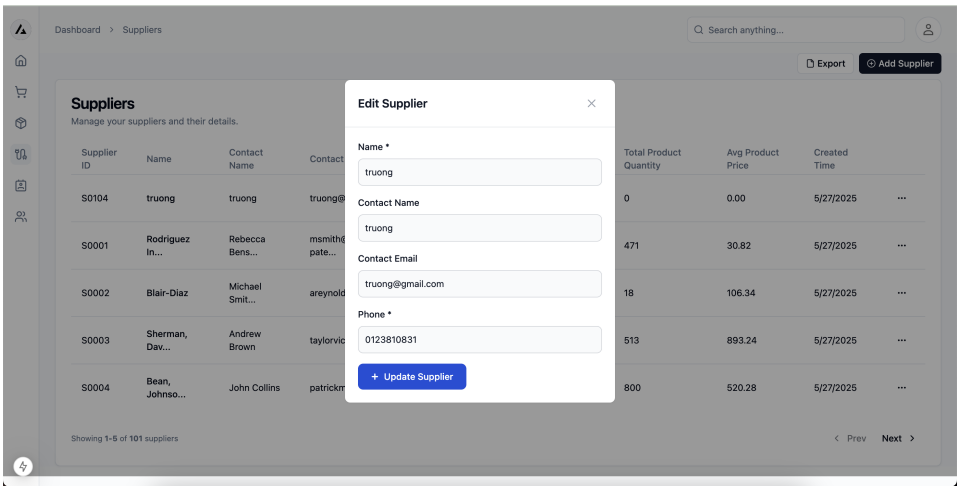


Figure 33: Edit supplier form

5.5.4 Super Search

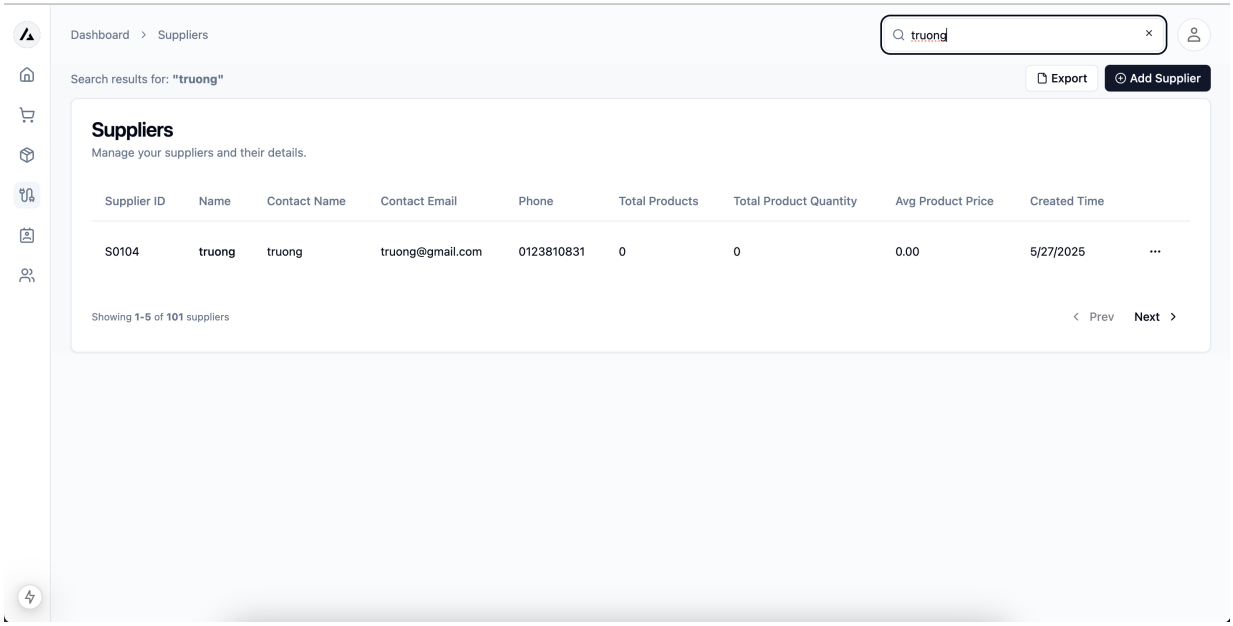


Figure 34: Super search - filtering by keyword 1

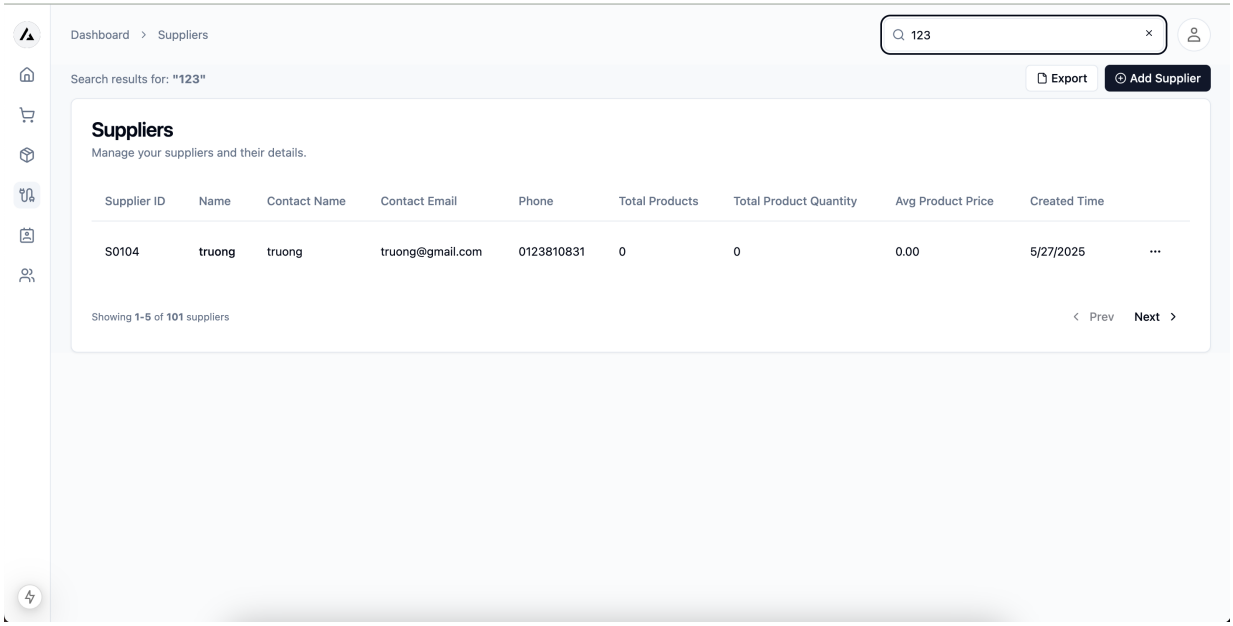


Figure 35: Super search - filtering by keyword 2

5.5.5 Supplier Docs

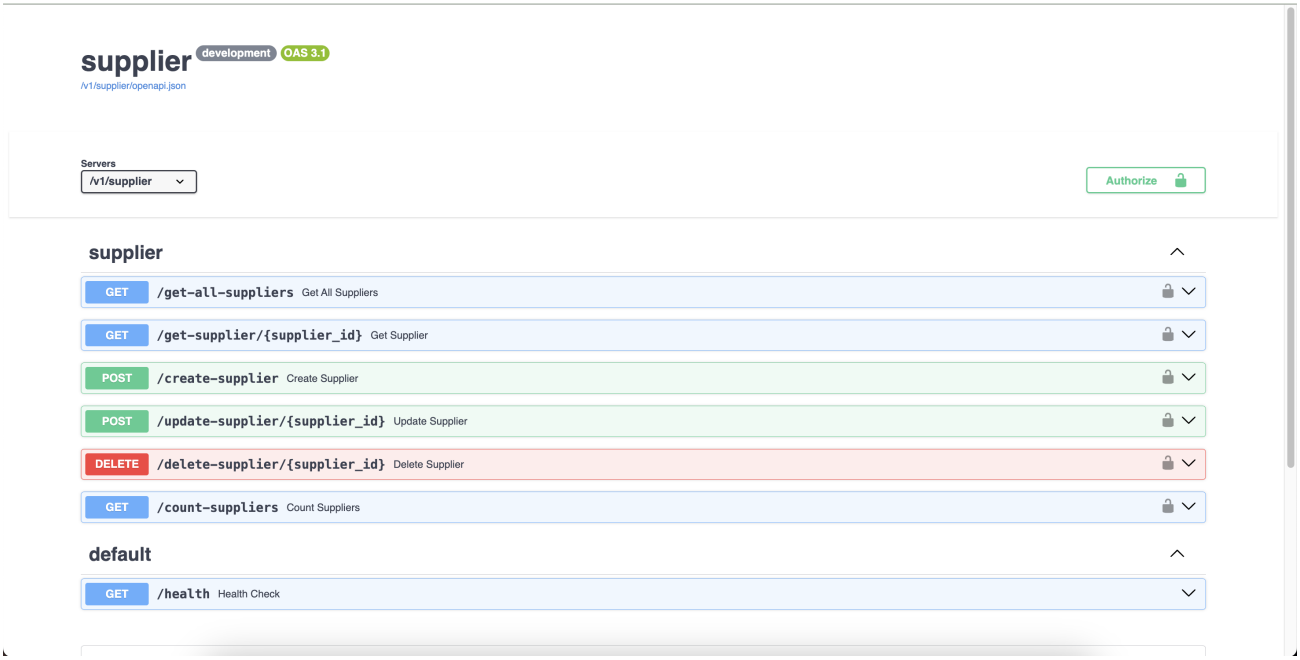


Figure 36: Supplier API endpoint documentation

5.5.6 Supplier Detail with Product

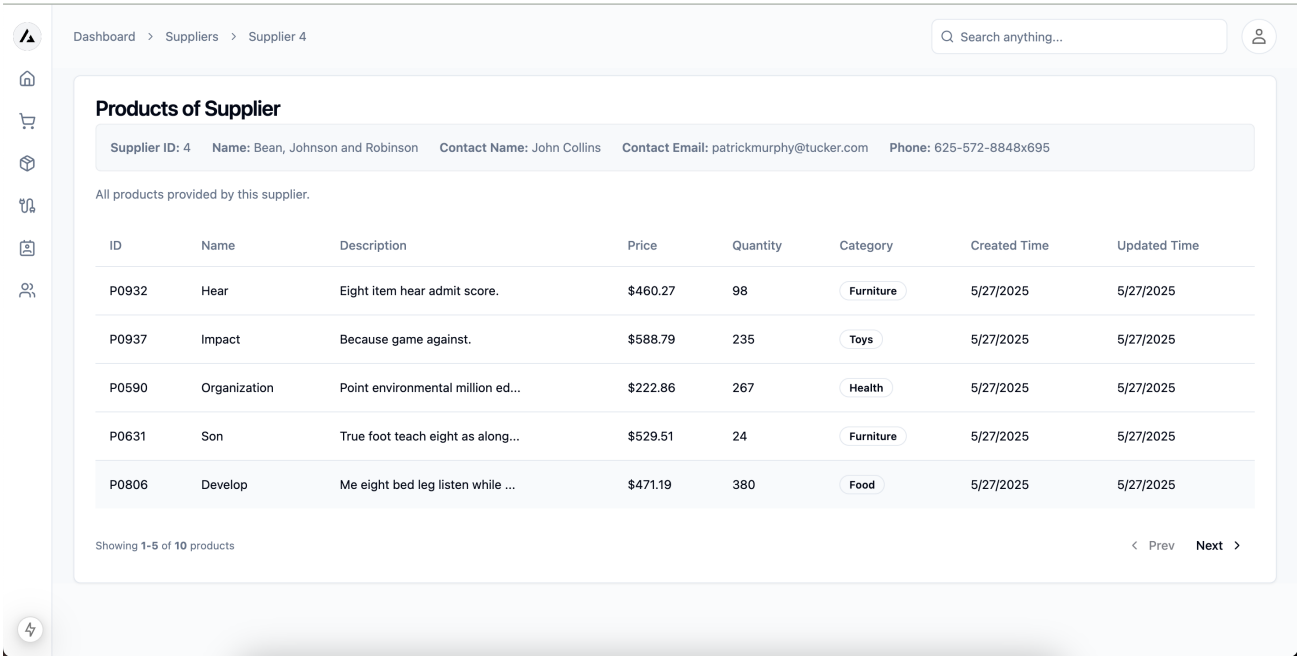


Figure 37: Supplier details with associated products

5.6 Order View

5.6.1 For Staff/Admin

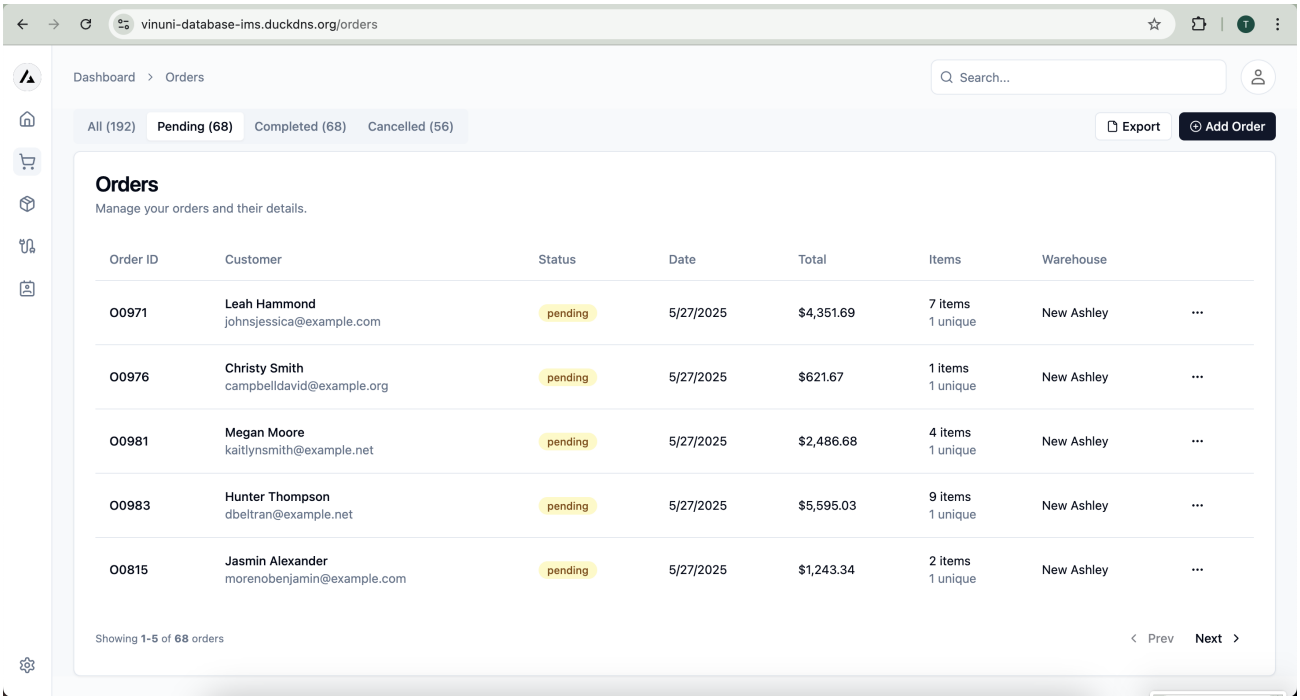


Figure 38: Pending orders view

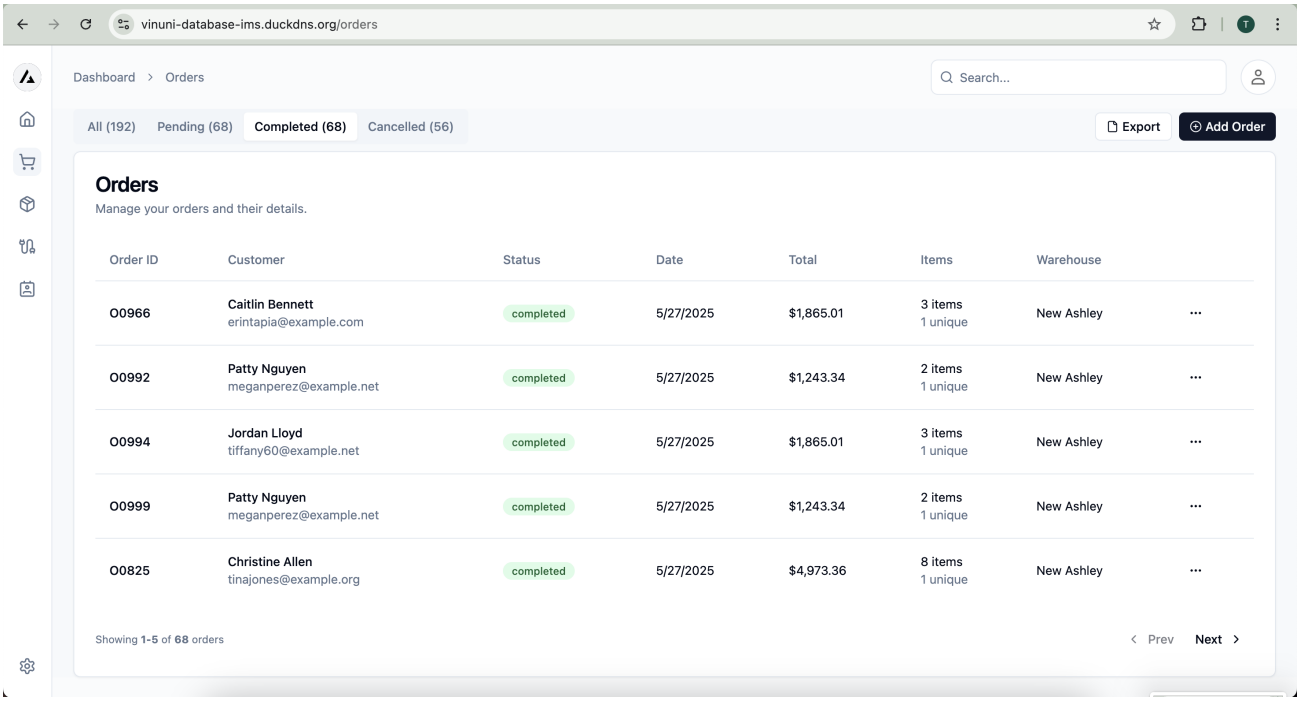


Figure 39: Successfully fulfilled orders

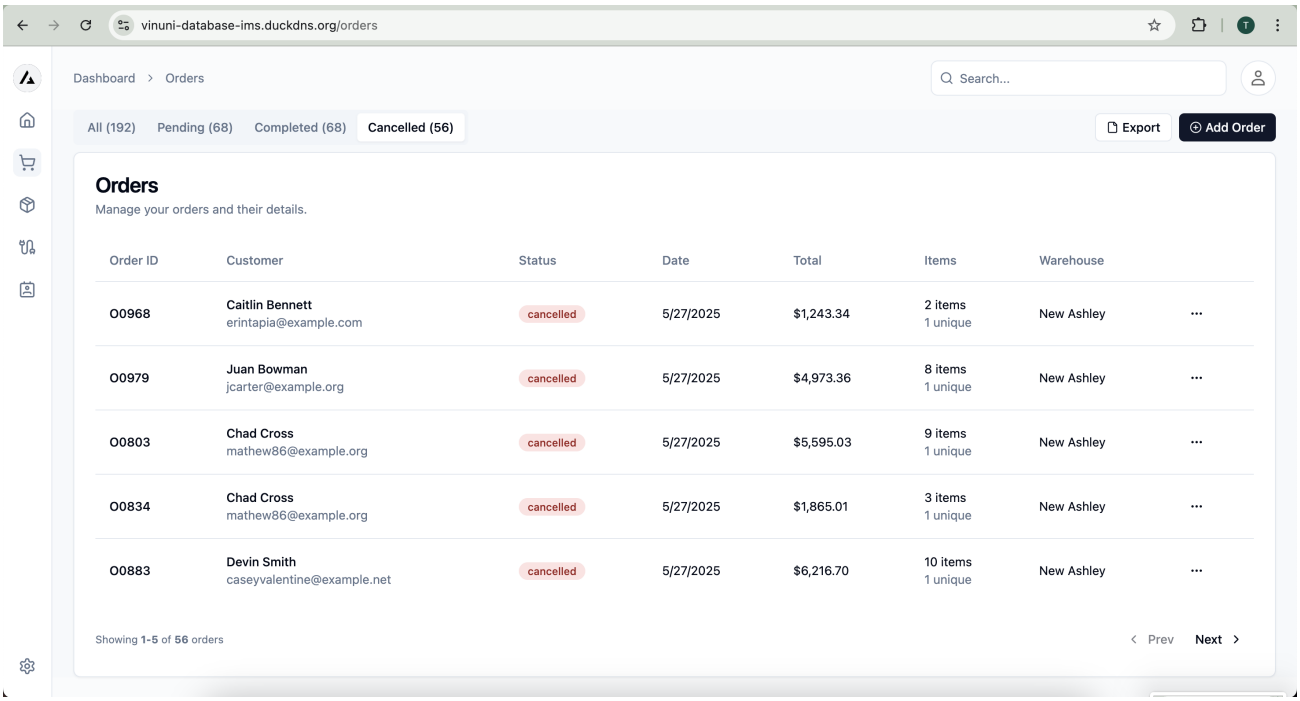


Figure 40: Cancelled orders

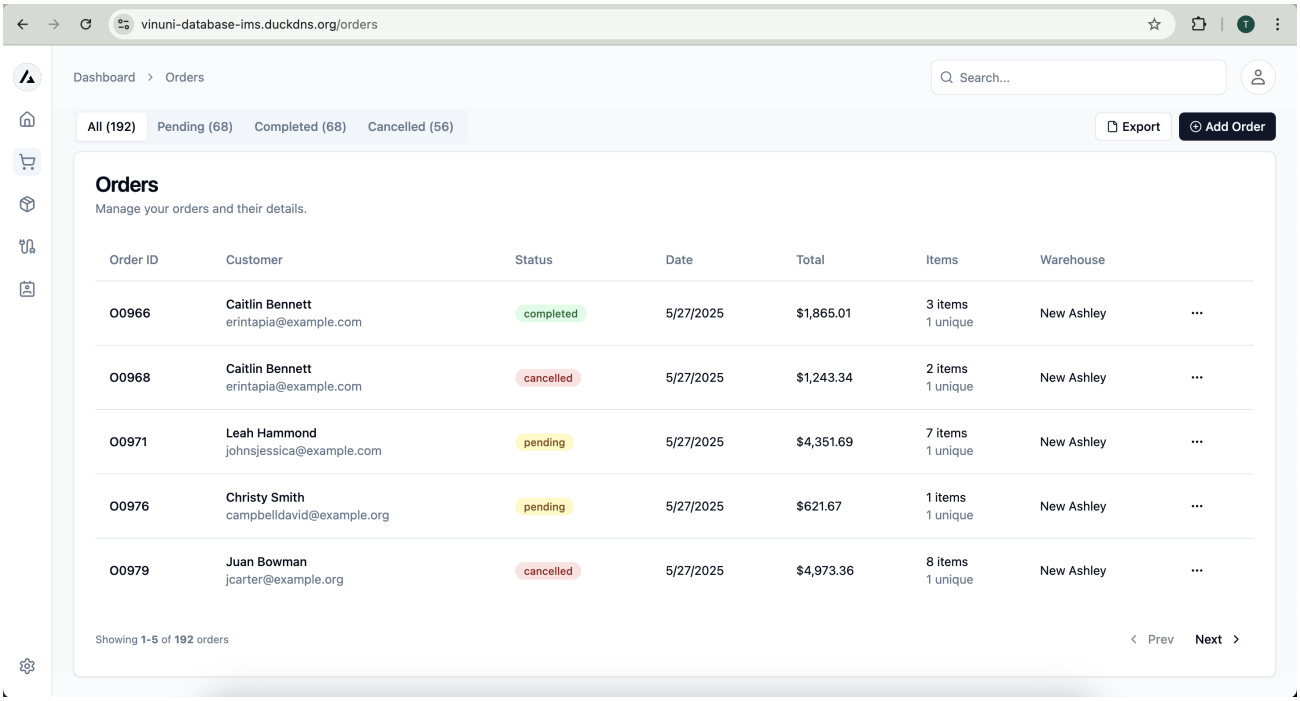


Figure 41: All orders overview

5.6.2 Order detail with order item

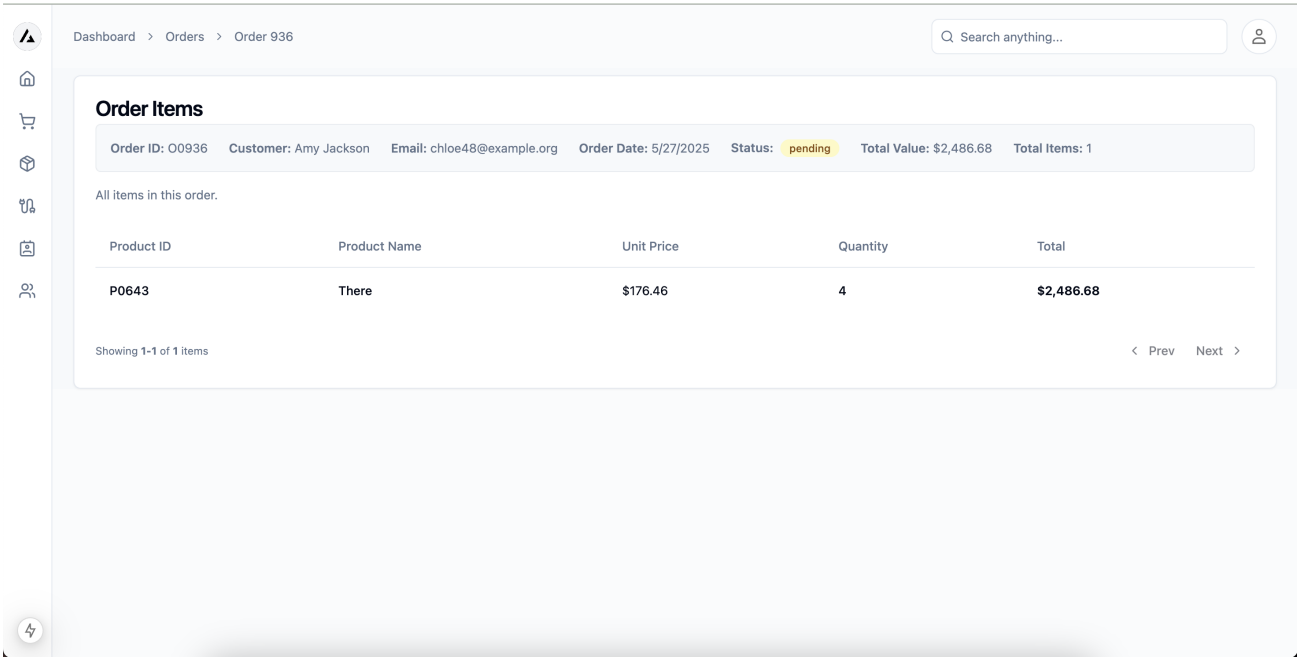


Figure 42: Order detail

6 Presentation & Material

Source code:

https://github.com/truongng201/Database-IMS-Project

All the related document and presentation slide will be at this link

https://github.com/truongng201/Database-IMS-Project/tree/main/docs