# Text Description automatically generated with low confidence

**American University of Beirut**

**Maroun Semaan Faculty of Engineering and Architecture**

**Department of Electrical and Computer Engineering**

Ecommerce Website Backend Service

By:

**Joseph Souaiby**

([jgs26@mail.aub.edu](mailto:jgs26@mail.aub.edu))

**Antoine Saade**

([ays13@mail.aub.edu](mailto:ays13@mail.aub.edu))

A REPORT

submitted to Dr. Iman Ghalayini in fulfillment of the requirements of the final project of the course EECE435L –Software Tools Lab on December 3rd, 2024

# Introduction

**Project Overview:** This project focuses on developing the backend web services for an e-commerce website. The system consists of four key services—Customers, Inventory, Sales, and Reviews—which will handle customer registration, product management, sales transactions, and product reviews respectively. These services are designed to interact with each other through API calls, providing a seamless and efficient user experience. The project is containerized using Docker to ensure consistent deployment across various environments. The goal is to build a robust, scalable, and secure backend system that supports an e-commerce platform's core functionalities, such as user management, product inventory, and order processing.

**Objectives:** The main objectives of this project are:

1. To develop four independent backend services for handling customers, inventory, sales, and reviews.
2. To ensure effective inter-service communication via API calls.
3. To utilize Docker for containerization and ensure the services run consistently across different environments.
4. To implement user authentication, error handling, and data validation to ensure the security and reliability of the system.
5. To perform thorough testing using Pytest and validate the system's functionality and performance.
6. To apply professional development practices, such as version control, documentation, and profiling, to ensure the project meets industry standards.
7. To integrate additional features like rate-limiting, logging, and secure configuration management for enhanced functionality and security.

# System Architecture

The system is composed of four backend services—Customers, Inventory, Sales, and Reviews—designed to provide the core functionality of an e-commerce platform. These services interact with each other to manage customer information, product inventory, sales transactions, and product reviews, while ensuring scalability, flexibility, and separation of concerns.

## Customers Service

Responsibilities: The Customers service manages all aspects of customer data, including account creation, management, and updating of personal details. It also handles wallet management, where customers can add or deduct funds for purchases. The service ensures secure customer registration, updates, and retrieves customer information as needed. Additionally, it handles account deletions and allows for querying customer details.

## Inventory Service

Responsibilities: The Inventory service manages the products available for sale. It is responsible for adding new products to the inventory, updating product information (such as price and description), and removing products when necessary. The service ensures that stock levels are properly maintained and allows for tracking of available items for sale. It provides functionality to check and update product availability, ensuring accurate listings on the platform.

## Sales Service

Responsibilities: The Sales service handles all transactions between customers and the platform. It ensures that products are purchased only when sufficient funds are available in the customer's wallet and when the product is in stock. The service processes the transaction by charging the customer’s wallet and updating the inventory to reflect the sale. It also tracks purchase history and maintains records of all transactions for customers.

## Reviews Service

Responsibilities: The Reviews service allows customers to leave feedback on products they have purchased. Customers can provide ratings and comments on products, helping other customers make informed decisions. The service supports moderating reviews to ensure quality and appropriateness of content. It also tracks reviews submitted by each customer and allows for retrieving all reviews related to a specific product.

## Communication Between Services

Each service communicates with others through API calls, ensuring that the system operates in an integrated and seamless manner. For instance, the Sales service communicates with the Customers service to verify available funds in the customer’s wallet and with the Inventory service to ensure product availability before processing a sale. The Reviews service interacts with both the Sales and Inventory services to ensure that only customers who have made a purchase can leave a review for a product.

## Docker Containerization

Each service is containerized using Docker to ensure isolated, consistent environments across development and production stages. Docker provides flexibility in deploying and scaling the services, while ensuring that dependencies and configurations are handled properly. The services are organized into a multi-container setup managed by Docker Compose, which enables streamlined orchestration and deployment.

# Implementation Details

**Service 1 - Customers**

**Functionality:** The Customers service is responsible for managing customer data, including registration, updating details, wallet management, and retrieving customer information.

**Key APIs:**

* **Register Customer (/register)**: Handles new customer registration, validating inputs like username, password, and other personal details.
* **Get All Customers (/customers)**: Retrieves a list of all registered customers.
* **Get Customer by Username (/customer/<username>)**: Fetches details for a specific customer.
* **Update Customer (/update/<username>)**: Updates personal information, such as name, age, or password.
* **Wallet Management**: Provides endpoints to charge (/charge/<username>) and deduct funds (/deduct/<username>) from a customer's wallet.

**Challenges Faced:**

* **Validation:** Ensuring inputs like usernames, passwords, and optional fields (e.g., age, gender) conform to business rules.
* **Username Uniqueness:** Preventing duplicate usernames during registration.
* **Data Consistency:** Proper handling of edge cases, such as invalid or missing fields in requests.

**Service 2 - Inventory**

**Functionality:** The Inventory service is used to manage goods, enabling administrators to add, update, retrieve, and delete inventory items.

**Key APIs:**

* **Add Item (/add\_item)**: Allows adding new items with fields like name, category, price, description, and stock.
* **Update Item (/update\_item/<name>)**: Updates the price, description, or stock of an item.
* **Deduct Stock (/deduct\_stock/<name>)**: Reduces the stock of an item after a purchase.
* **Get All Items (/get\_items)**: Retrieves a list of all items in the inventory.
* **Get Item by Name (/get\_item/<name>)**: Fetches detailed information for a specific item.
* **Delete Item (/delete\_item/<name>)**: Removes an item from the inventory.

**Challenges Faced:**

* **Category Validation:** Ensuring categories (e.g., "Electronics", "Food") match predefined options.
* **Stock Management:** Handling cases where stock may become negative or insufficient for deductions.
* **Error Handling:** Returning appropriate errors when items are not found or validations fail.

**Service 3 - Sales**

**Functionality:** The Sales service processes purchases made by customers and maintains a historical record of transactions.

**Key APIs:**

* **Display Available Goods (/display\_goods)**: Lists all goods in stock with their names and prices.
* **Get Good Details (/goods/<name>)**: Fetches detailed information for a specific item.
* **Make Purchase (/purchase)**: Processes a purchase, deducting funds from the customer's wallet and updating inventory stock.
* **Get Purchase History (/purchase\_history/<username>)**: Retrieves all past purchases made by a specific customer.

**Challenges Faced:**

* **Multi-Step Transactions:** Ensuring wallet deduction, stock updates, and sale recording are atomic (completed as a single logical unit).
* **Error Scenarios:** Handling edge cases, such as insufficient wallet balance, low stock, or invalid purchase requests.
* **Historical Data Management:** Maintaining a detailed and easily retrievable history of transactions.

**Service 4 - Reviews**

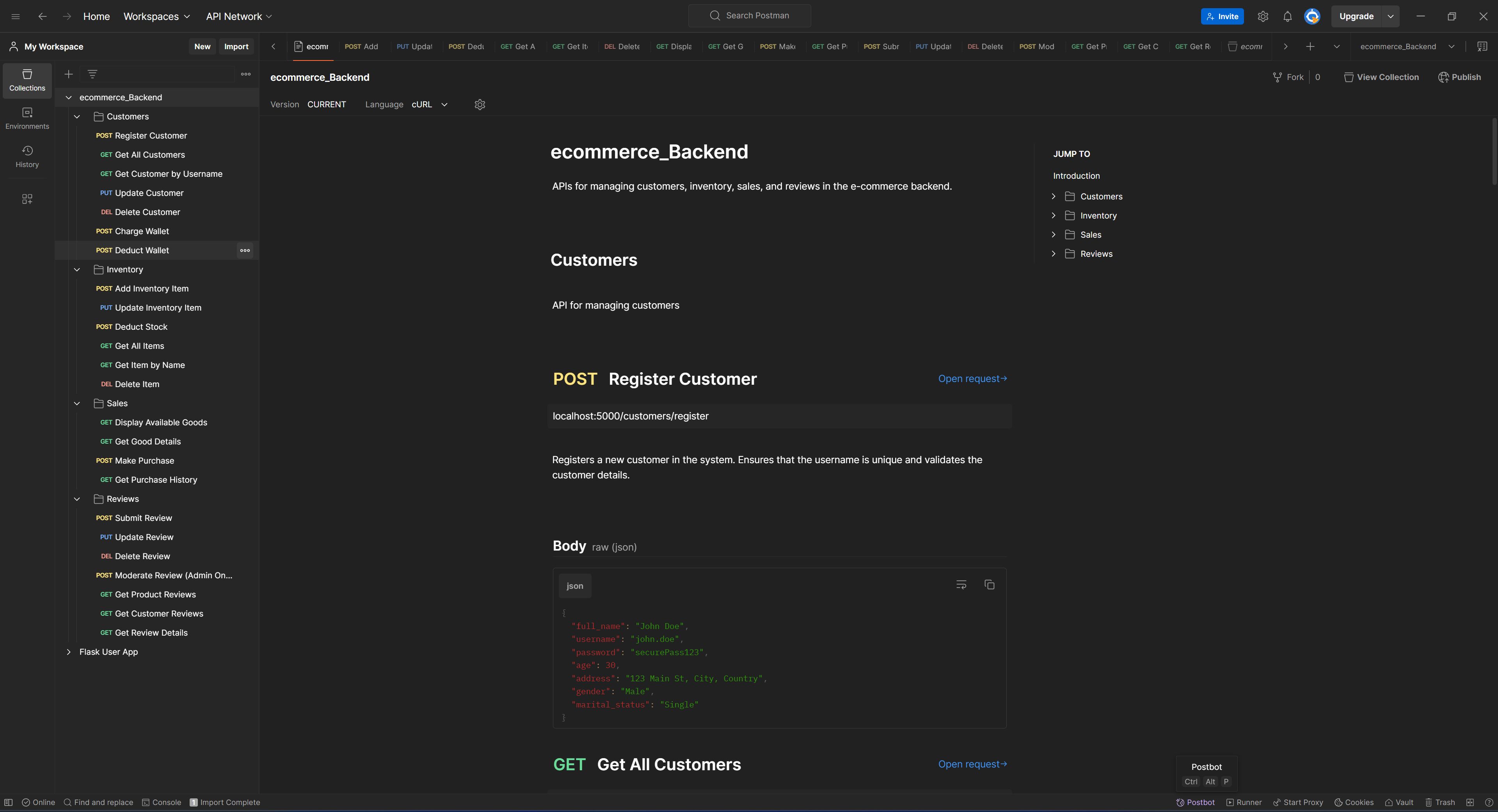
**Functionality:** The Reviews service allows customers to submit, update, retrieve, and moderate reviews for inventory items.

**Key APIs:**

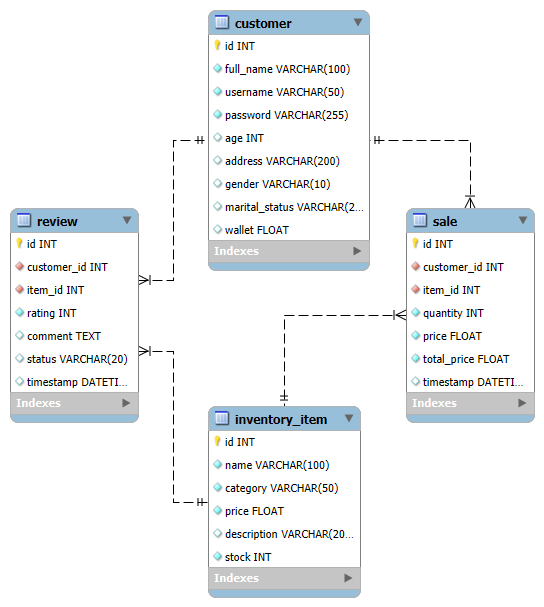
* **Submit Review (/submit)**: Enables customers to submit a new review for a product. Reviews initially have a "pending" status.
* **Update Review (/update/<review\_id>)**: Allows customers to modify their review's rating or comment. The status reverts to "pending" after an update.
* **Delete Review (/delete/<review\_id>)**: Permits customers to delete their own reviews.
* **Get Product Reviews (/product\_reviews/<item\_name>)**: Retrieves all approved reviews for a specific product.
* **Get Customer Reviews (/customer\_reviews/<username>)**: Fetches all reviews submitted by a specific customer.
* **Moderate Review (/moderate/<review\_id>)**: Allows administrators to approve, flag, or delete reviews.

**Challenges Faced:**

* **Ownership Validation:** Ensuring only the review author can update or delete a review.
* **Review Moderation:** Implementing a process for administrators to review and approve or reject submissions.
* **Performance:** Optimizing retrieval of reviews for products with many submissions.

**Now for the full postman collection, you will find on our GitHub a file called “ecommerce\_Backend.postman\_collection.json”. Save it on your laptop and import it into postman and you will have our collection. Below is a screenshot of the collection on my postman Desktop app. You should get the same when you import it and will be able to test all API endpoints from there. You will find the full documentation there too.  
  
**

# Database Design



# Error Handling and Validation

**Error Management**

**Description:** The system incorporates robust error handling mechanisms to ensure that invalid requests, system issues, or unexpected conditions are addressed gracefully, providing meaningful feedback to users and developers.

**Key Error Handling Techniques:**

1. **HTTP Status Codes:**
   * **200 OK**: For successful operations.
   * **201 Created**: For successful resource creation (e.g., customer registration, review submission).
   * **400 Bad Request**: For validation errors or invalid input data.
   * **403 Forbidden**: For access violations (e.g., trying to modify or delete another customer's review).
   * **404 Not Found**: For resources that do not exist (e.g., customer, item, or review).
   * **500 Internal Server Error**: For unexpected system errors.
2. **Error Messages:**
   * The system returns JSON responses with detailed error messages for client-side issues. For example:

{

"error": "Customer not found."

}

* + For validation errors, multiple errors are returned as a list:

{

"errors": [

"Full name must be at least 3 characters long.",

"Password must be at least 8 characters long."

]

}

1. **Database Error Handling:**
   * The system uses try-except blocks (or equivalent) to catch database-related exceptions, such as connection issues or unique constraint violations, and responds with appropriate HTTP status codes and messages.
2. **Input Missing/Error Cases:**
   * Missing fields (e.g., username or item\_name) or invalid data types result in a 400 Bad Request response with specific details about what is missing or invalid.

**Validation**

**Description:** Validation is implemented across all services to ensure data integrity and prevent invalid inputs from entering the system. Validation is applied at both the client request level and within the backend logic.

**Types of Validation:**

**1. Customers Service**

* **Registration (/register):**
  + full\_name: Must be at least 3 characters long.
  + username: Must contain only letters, numbers, and . or \_ or -. Unique across all users.
  + password: Must be at least 8 characters long.
  + age: Must be a non-negative integer.
  + gender: Must be one of "Male", "Female", or "Other" (case-insensitive).
  + marital\_status: Must be one of "Single", "Married", "Divorced", or "Widowed" (case-insensitive).
* **Wallet Management (/charge/<username>, /deduct/<username>):**
  + amount: Must be a positive number.
  + Sufficient funds are validated before deducting money.

**2. Inventory Service**

* **Adding and Updating Items (/add\_item, /update\_item/<name>):**
  + name: Must be at least 3 characters long.
  + category: Must be one of "Food", "Clothes", "Accessories", or "Electronics" (case-insensitive).
  + price: Must be a positive number.
  + stock: Must be a non-negative integer.
* **Deducting Stock (/deduct\_stock/<name>):**
  + quantity: Must be a positive integer.
  + Sufficient stock is validated before processing the deduction.

**3. Sales Service**

* **Making a Purchase (/purchase):**
  + username: Must belong to a valid, registered customer.
  + item\_name: Must refer to a valid, existing item in the inventory.
  + quantity: Must be a positive integer.
  + **Validation Checks:**
    - Sufficient funds in the customer's wallet.
    - Sufficient stock available for the item.

**4. Reviews Service**

* **Submitting Reviews (/submit):**
  + username: Must belong to a valid customer.
  + item\_name: Must refer to a valid inventory item.
  + rating: Must be an integer between 1 and 5.
  + comment: Optional but must be a valid string if provided.
* **Updating Reviews (/update/<review\_id>):**
  + Validation ensures that only the review author can update the review.
  + rating: Must be an integer between 1 and 5.
  + comment: Must be valid if provided.
* **Moderating Reviews (/moderate/<review\_id>):**
  + The action field must be one of "approve", "flag", or "delete".

**Validation Locations:**

1. **Request-Level Validation:**
   * All incoming requests are validated using custom Python functions before processing. For example, validate\_customer\_data in the customers.py file ensures registration data is correct.
2. **Database-Level Validation:**
   * Constraints like unique usernames and non-nullable fields are enforced in the database schema.
3. **Error Feedback:**
   * Detailed validation errors are returned to the client for incorrect or incomplete requests.

# 6. Testing

**Testing Strategy**

The application is tested using the following approaches:

**1. Unit Testing**

* Focuses on testing individual functions and methods in isolation.
* Each function (e.g., adding an item, submitting a review) is tested with valid and invalid inputs.

**2. Integration Testing**

* Tests how various components (e.g., Customers, Inventory, Sales, and Reviews) work together.
* Example: Testing how a purchase affects both customer wallet and inventory stock.

**3. Database Testing**

* Uses an in-memory SQLite database for testing purposes.
* Ensures data consistency and integrity (e.g., no duplicate usernames, no negative stock).

**4. Test Fixtures**

* Prepares test data (e.g., customers, inventory items) before running tests and cleans up after.

**5. Tools Used**

* **Pytest**: A Python testing framework.
* **SQLite (in-memory)**: Provides a lightweight database for testing without affecting production data.

**Test Cases**

**1. Customers Service (test\_customers.py)​1**

* **Register Customer**
  + Valid registration → Status: 201 Created
  + Duplicate username → Status: 400 Bad Request
  + Missing fields → Status: 400 Bad Request
  + Invalid gender, age, or marital status → Status: 400 Bad Request
* **Customer Retrieval**
  + Retrieve all customers → Status: 200 OK
  + Retrieve single customer → Status: 200 OK or 404 Not Found
* **Wallet Management**
  + Charge wallet → Status: 200 OK
  + Deduct wallet (sufficient funds) → Status: 200 OK
  + Deduct wallet (insufficient funds) → Status: 400 Bad Request

**2. Inventory Service (test\_inventory.py)​2**

* **Add Item**
  + Valid data → Status: 201 Created
  + Duplicate item name → Status: 400 Bad Request
  + Invalid data (e.g., negative stock, invalid category) → Status: 400 Bad Request
* **Update Item**
  + Valid update → Status: 200 OK
  + Nonexistent item → Status: 404 Not Found
* **Deduct Stock**
  + Valid deduction → Status: 200 OK
  + Insufficient stock → Status: 400 Bad Request
* **Retrieve Items**
  + All items → Status: 200 OK
  + Single item → Status: 200 OK or 404 Not Found
* **Delete Item**
  + Valid deletion → Status: 200 OK
  + Nonexistent item → Status: 404 Not Found

**3. Reviews Service (test\_reviews.py)​3**

* **Submit Review**
  + Valid data → Status: 201 Created
  + Invalid rating, nonexistent customer/item → Status: 400/404 Bad Request
* **Update Review**
  + Valid update by author → Status: 200 OK
  + Invalid author → Status: 403 Forbidden
  + Nonexistent review → Status: 404 Not Found
* **Delete Review**
  + Valid deletion by author → Status: 200 OK
  + Invalid author → Status: 403 Forbidden
* **Retrieve Reviews**
  + Product reviews → Status: 200 OK or 404 Not Found
  + Customer reviews → Status: 200 OK or 404 Not Found
* **Moderate Review**
  + Approve, or delete → Status: 200 OK
  + Invalid action → Status: 400 Bad Request
* **Flag Review**
  + Flag a review → Status: 200 OK or 404 Not Found
* **get Flagged Reviews**
  + View all flagged reviews → Status: 200 OK

**4. Sales Service (test\_sales.py)​4**

* **Display Goods**
  + Valid request → Status: 200 OK
* **Get Good Details**
  + Valid item → Status: 200 OK
  + Nonexistent item → Status: 404 Not Found
* **Make Purchase**
  + Valid purchase → Status: 200 OK
  + Insufficient funds/stock → Status: 400 Bad Request
  + Nonexistent customer/item → Status: 404 Not Found
* **Purchase History**
  + Existing purchases → Status: 200 OK
  + No purchases → Status: 200 OK (empty array)
  + Nonexistent customer → Status: 404 Not Found

**Pytest Results Summary**

1. **Execution:**
   * Tests are executed using pytest from the terminal:

pytest --tb=short

1. **Results:**
   * **Total Tests:** Count of all unit and integration tests across services.
   * **Passed:** All valid test cases.
   * **Failed:** None (if properly implemented).
   * **Skipped:** Tests deliberately skipped due to incomplete features.
2. **Example Summary:**

A screenshot of a computer program

Description automatically generated

# 7. Deployment and Integration

**Docker Setup:**

1. **Dockerfile Creation**: I started by creating a Dockerfile for each service. The Dockerfile defines the environment for the service, including the base image, dependencies, and the commands to run the service. For each service, I used Python’s official image as the base, then added the necessary Python dependencies from a requirements.txt file.
2. **Docker Compose**: To manage the multiple services and their dependencies, I created a docker-compose.yml file. This file defines the services, how they interact, and sets up the necessary networks and volumes for persistent storage, if needed. It also manages the inter-service communication. I made sure that the services could communicate with each other by linking them via Docker’s internal networking.
3. **Networking**: I ensured that all services were able to communicate with each other through Docker’s default bridge network. For example, the Customers service communicates with the Inventory service over HTTP by making API calls, so they need to be able to resolve each other’s names (e.g., customers, inventory).
4. **Testing and Debugging**: Once the services were containerized and set up with Docker Compose, I tested the setup by building and running the containers using:

I checked if all the services were running correctly and if the inter-service communication worked as expected by inspecting logs and testing API endpoints.

1. **Volume Mapping**: For persistence and to avoid losing data when containers are stopped, I mapped necessary volumes (such as databases or uploaded files) to the host machine. This ensured that data could persist between container restarts.
2. **Security and Performance**: I made sure to follow the best practices in container security, such as running services with a non-root user inside the container, minimizing the size of the images, and ensuring that sensitive information (e.g., credentials) was securely stored.

By containerizing the services with Docker, I could easily scale, test, and deploy the application while ensuring that the environment remained consistent across different systems. It also simplifies the deployment process, allowing me to integrate all the services smoothly and ensure that they work well together in a production-like environment.

**Keep in mind I changed the API base\_url to match the ones of the containers. I have 5 containers running: the 4 services and the mysql\_container.**

# 8. Documentation and Profiling

## 8.1. Documentation

You can find the html file I created with Sphinx in **build/html/index.html**

This ducmentation was generated from the docstring in my code. You can check the conf.py and index.rst to further see what I did for the “make html” command to produce that. Find below a screenshot of the run that created the HTML:

A black screen with white text

Description automatically generated

## 8.2 Profiling

# 10. Docker and Images

To package the applications and their dependencies into containers for consistent and efficient deployment across various environments, I followed these steps:

**1. Dockerfile Creation:**

I started by creating Dockerfile for each of the four services in my backend system (Customers, Inventory, Sales, and Reviews). The Dockerfile specifies the environment that each service needs to run, including the application code and any dependencies.

**2. Dependency Management:**

To ensure that the applications run consistently across various environments, I listed all required Python packages in a requirements.txt file. This file was copied into the Docker image during the build process.

**3. Docker Compose for Multi-Service Setup:**

I then used Docker Compose to handle the deployment of multiple services together. The docker-compose.yml file defines the different services (Customers, Inventory, Sales, and Reviews) and their interactions.

This file makes it easy to bring up the services together, with each one depending on a database container (db) that they interact with.

**4. Building and Running Containers:**

I built and ran the containers using the following commands:

docker-compose up --build

This command builds the Docker images from the Dockerfile in each service directory, creating containers for each service. Once the containers are built, they run according to the specifications in the docker-compose.yml file.

**Here is a screenshot of the containers running on Docker after I run docker-compose up –build**

A black screen with white text

Description automatically generated

**I even added a health\_check and depends\_on in the docker-compose.yml file to make sure of the order of the containers since they depend on each other.**

**This way my run for the screenshot above:**

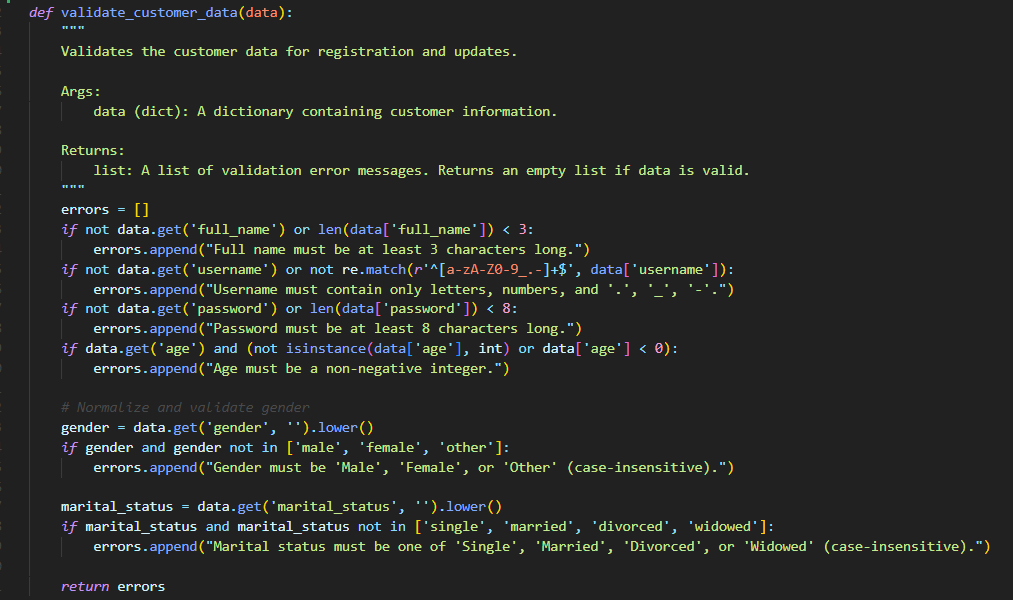
**A screen shot of a computer

Description automatically generated**

**You can see the 5 containers being created and running**

# 11. Validation and Sanitization

Validation and sanitization were implemented for user inputs in all our services to ensure data integrity and security. This prevents SQL injection and ensures that inputs meet the required format. Inputs like review text, ratings, age, usernames, product IDs and many more are validated for correctness and sanitized to avoid malicious payloads.

Examples of validation is this function in the customer service that validates the input needed to register a customer. It is discussed in detail previously.

For sanitization and preventing SQL injection and other security issues we used SQLAlchemy ORM for all queries, and to sanitize inputs in all the services.

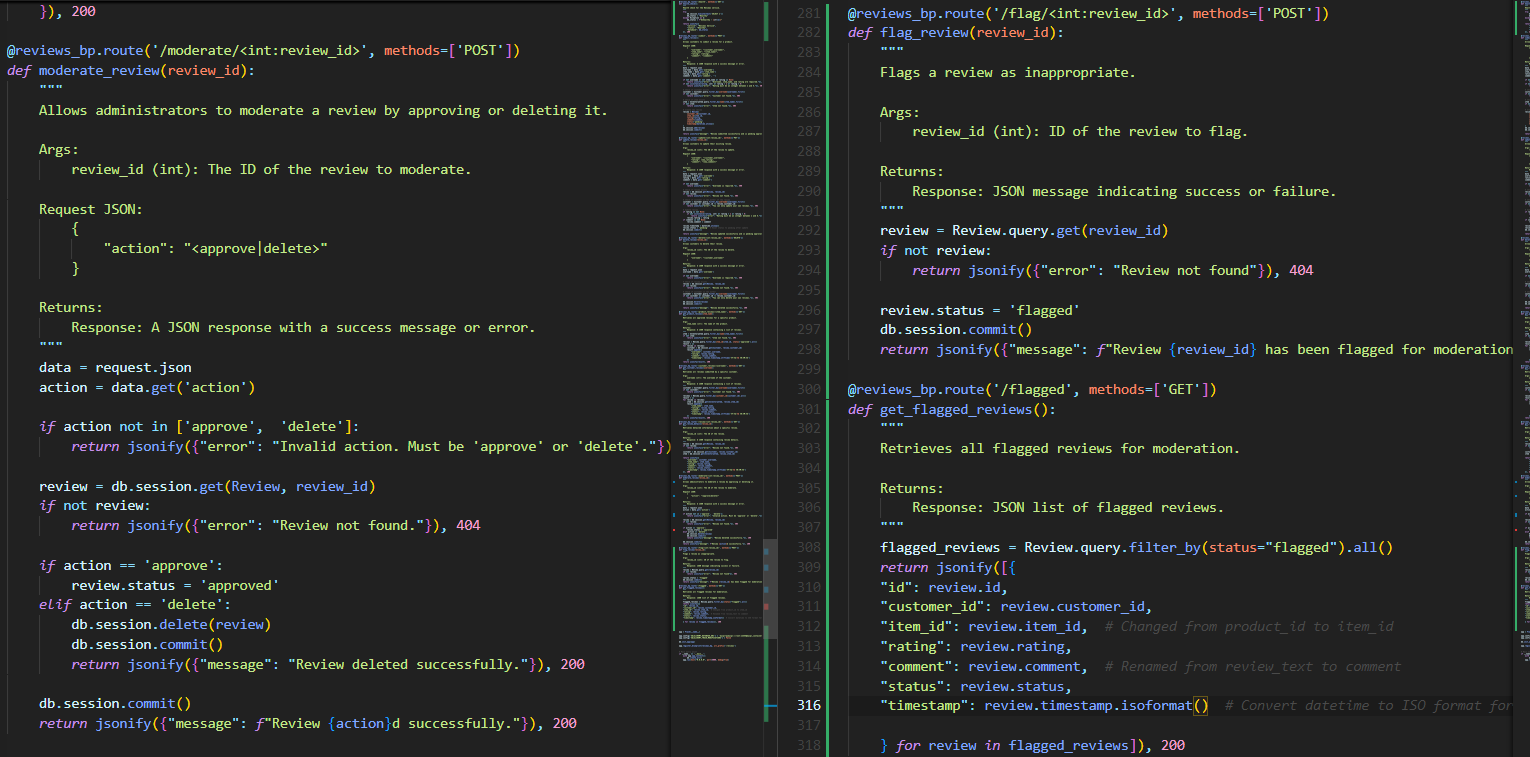
The ORM automatically parameterizes queries, ensuring inputs are safely escaped. For example, the filter\_by method in the customer service ensures that queries such as filter\_by(username=username) are executed in a secure and convenient wayA black screen with colorful text

Description automatically generated

This combination of validation and ORM ensures robust protection against SQL injection and data integrity issues.

# 13. Moderation

To handle inappropriate reviews effectively, we implemented a comprehensive moderation system using three key routes: **flag review**, **get flagged reviews**, and **moderate review** as shown below.



The flag review route allows users to flag a review as inappropriate by sending a POST request with the review ID. The system checks if the review exists and, if valid, marks it as flagged by updating the status column in the database. This enables flagged reviews to be identified for administrative review.

Administrators can view all flagged reviews using the get flagged reviews route, which returns a list of flagged reviews with detailed information, including the review's rating and timestamp. Finally, the moderate review route allows administrators to either approve a flagged review (marking it as approved) or delete it entirely. This process streamlines moderation by separating user-initiated flagging from administrative actions, ensuring inappropriate reviews are handled quickly and efficiently. The implementation ensures data integrity and leverages SQLAlchemy ORM for secure database interactions, preventing SQL injection and other security risks.

# 14. Additional Professional Tasks

1. Rate Limiting and Throttling: Apply rate-limiting strategies for APIs to prevent abuse or overload of individual services.

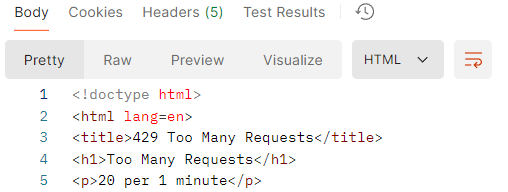
We used the Flask-Limiter library to implement rate limiting, we configured the limiter with global default limits and the get\_remote\_address function to identify clients by IP – this was done for every service. Then we applied specific time limits to the different services as shown below. This prevents overload and abuse of API endpoints and ensures stability.

* **Global Default Limit**:
  + The default\_limits=["25 per minute"] configuration applies to all endpoints within the specific Flask app, unless specified otherwise .
  + Any route accessed more than 25 times in a minute by the same client (IP address) will return a 429 Too Many Requests error.
* **Unified Limit for All Routes**:
  + Whether a user accesses /register, /customers, or /delete/<username>, all requests count toward the same limit in this case.

An example of that in code: A computer screen shot of a black screen

Description automatically generated

This is done for all the different services with different time limits. In this case the limit is 20 per min and when we exceed it we get the following



1. Security Practices: Explain measures taken to prevent common vulnerabilities (e.g., SQL injection).

The application leverages **SQLAlchemy ORM** for all database interactions. The ORM automatically parameterizes queries, ensuring user inputs are safely escaped and preventing malicious SQL injection attempts.

Instead of concatenating raw SQL strings, queries like filter\_by() were used

This ensures that even if malicious inputs are provided, they are treated as data rather than executable SQL.

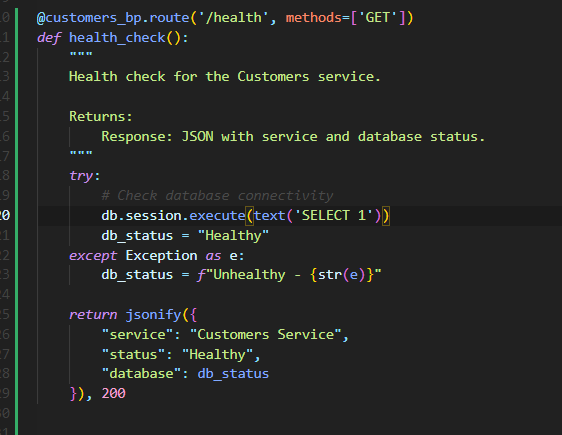
Comprehensive validation is performed for all user inputs to ensure they meet expected formats and constraints.

For instance, fields like username, age, and rating are validated to ensure they are of the correct type and within valid ranges

To prevent abuse and protect against Denial-of-Service (DoS) attacks, **Flask-Limiter** was implemented across all routes with global and service-specific rate limits. This ensures that users cannot overwhelm the system with excessive requests.

By combining these practices, the application ensures that user inputs are validated, and malicious queries are neutralized, thus significantly reducing the risk of common vulnerabilities and enhancing the overall security of the system.

1. Health Checks: Add health-check APIs for all services to monitor their availability and performance

To implement health checks, a /health endpoint was added to each service's blueprint to monitor service availability and database connectivity. The endpoint performs a lightweight query (SELECT 1) using SQLAlchemy's text() function to ensure the database is reachable and returns a JSON response indicating the service and database status.An example of the endpoint added (this was added to the inventory service)

This enables quick identification of service or database issues, allowing us to monitor their availability and performance.

This is the output after checking the health of the inventory service.