E-Commerce Website

By: Omar Kandil,

Sharafeddine Sharafeddine

Date: December 2, 20

|  |
| --- |

**2. Table of Contents**

 Title Page (Page: 1, Responsible: Omar)

 Table of Contents (Page: 2, Responsible: Omar)

 Introduction (Page: 4, Responsible: Omar)

* Project Overview (Page: 4, Responsible: Omar)
* Objectives (Page: 5, Responsible: Omar)

 System Architecture (Page: 5, Responsible: Sharafeddine)

 Implementation Details

* Service-Specific Implementation
  + Customers (Page: 7, Responsible: Sharafeddine)
  + Inventory (Page: 9, Responsible: Omar)
  + Sales (Page: 11, Responsible: Sharafeddine)
  + Reviews (Page: 12, Responsible: Omar)
* API Documentation (Page: 7-12 + Appendix, Responsible: Omar)

 Database Design (Page: 14, Responsible: Sharafeddine)

* Schema Diagram (Page: 14, Responsible: Sharafeddine)

 Error Handling and Validation (Page: 16, Responsible: Sharafeddine and Omar)

* Error Management (Page: 16, Responsible: Sharafeddine and Omar)
* Validation (Page: 16, Responsible: Sharafeddine and Omar)

 Testing (Page: 17, Responsible: Sharafeddine and Omar)

* Testing Strategy (Page: 17, Responsible: Sharafeddine and Omar)
* Test Cases (Page: 17, Responsible: Sharafeddine and Omar)

 Deployment and Integration (Page: 19, Responsible: Sharafeddine)

* Docker Setup (Page: 19, Responsible: Sharafeddine)

 Documentation and Profiling (Page: 24, Responsible: Sharafeddine and Omar)

* Documentation (Page: 24, Responsible: Sharafeddine and Omar)
* Performance Profiling (Page: 25, Responsible: Omar)

 GitHub and Version Control (Page: 27, Responsible: Sharafeddine and Omar)

* Repository Links (Page: 27, Responsible: Sharafeddine and Omar)

 Docker and Images (Page: 29, Responsible: Sharafeddine)

* Packaging and Deployment (Page: 29, Responsible: Sharafeddine)

 Validation and Sanitization (Page: 30, Responsible: Sharafeddine and Omar)

 User Authentication (Page: 32, Responsible: Sharafeddine and Omar)

 Moderation (Page: 34, Responsible: Omar)

 Additional Professional Tasks (Page: 35, Responsible: Sharafeddine and Omar)

 References (Page: 37, Responsible: Sharafeddine and Omar)

 Appendices (Page: 38, Responsible: Omar)

**3. Introduction**

**Project Overview:**

The E-Commerce API Project is a modular application that serves as the backend for an online shopping platform. This project provides essential functionalities to manage the lifecycle of e-commerce operations, including customer account creation and management, inventory tracking, sales transaction processing, and review moderation. It leverages the Flask web framework for its lightweight and extensible architecture, SQLite for data storage, and Docker for containerization and consistent deployment. The project is structured into distinct services, each responsible for a specific domain of the e-commerce platform, ensuring a separation of concerns and scalability. The application is designed to facilitate seamless and secure interactions between users and the platform, offering comprehensive API endpoints that handle all critical operations.

**Objectives**:

* + Customer Management: Create and manage customer accounts, implement secure authentication, and support additional functionalities such as wallet management and profile updates.
  + Inventory Management: Allow for the addition, updating, and deletion of products in the inventory. Enable accurate stock tracking to ensure that the platform reflects the availability of products in real-time.
  + Sales Processing: Handle purchase transactions efficiently while maintaining a detailed history of sales. Update inventory levels dynamically to reflect sold items and provide comprehensive sales reports.
  + Review System: Allow customers to submit, edit, delete, and view product reviews while implementing a moderation mechanism to handle inappropriate content flagged by users or administrators.
  + Secure and Reliable Operation: Implement robust input validation and sanitization mechanisms to prevent vulnerabilities such as SQL injection and cross-site scripting. Ensure only authenticated and authorized users can access specific functionalities.
  + Documentation and Profiling: Provide developer-friendly documentation for all APIs, ensuring clarity and ease of use. Use profiling tools to analyze system performance and optimize code for better efficiency.
  + Testing and Quality Assurance: Establish a comprehensive testing strategy encompassing unit, integration, and system testing. Ensure that the system functions reliably under various scenarios and edge cases.
  + Efficient Deployment and Integration: Use Docker to containerize services, ensuring compatibility and ease of deployment across development, staging, and production environments.

This project aims to offer a robust and secure solution for e-commerce management, emphasizing modularity, scalability, and reliability. It serves as a practical demonstration of applying software engineering principles to solve real-world problems while maintaining a developer-friendly design for future enhancements.

**4. System Architecture**

**Service Description:**

The E-Commerce API Project is organized into modular services, each responsible for handling a specific aspect of the e-commerce platform. These services ensure a clean separation of concerns, making the system scalable and maintainable. Below are detailed descriptions of each service:

1. **Customers Service**

Responsibilities:

* Manage user accounts, including registration, authentication, and profile updates.
* Implement wallet functionality for storing and using account credits.
* Securely handle password storage using Flask-Bcrypt for hashing.
* Provide endpoints for user-related queries and updates.

Key Features:

* User registration and login.
* Password recovery mechanism.
* Wallet balance retrieval and updates.

2. **Inventory Service**

Responsibilities:

* Manage product data, including adding, updating, and deleting items.
* Track stock levels to ensure real-time inventory accuracy.
* Support searching and filtering products based on various attributes like category, price range, and availability.

Key Features:

* Endpoint to add new products to the inventory.
* Update product details and stock levels dynamically.
* Retrieve lists of products with sorting and filtering options.

3. **Sales Service**

Responsibilities:

* Process purchase transactions, linking customers to their purchases.
* Update inventory levels after successful sales transactions.
* Maintain a history of sales for customer reference and reporting.

Key Features:

* Endpoint to initiate and complete a sale.
* Generate receipts or order confirmation.
* Provide sales analytics for administrators.

4. **Reviews Service**

Responsibilities:

* Allow customers to submit, update, delete, and retrieve reviews for products.
* Enable review moderation, including flagging inappropriate content by users or administrators.
* Ensure reviews are tied to authenticated customers to prevent spam.

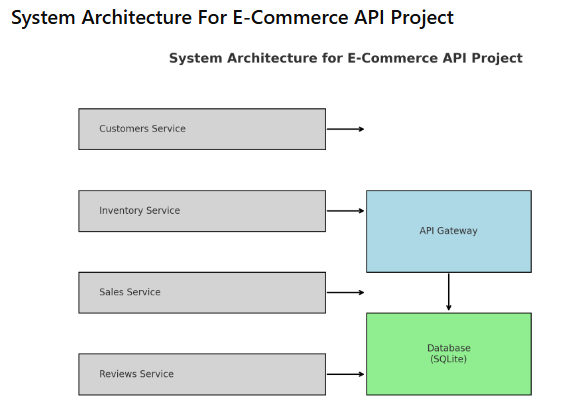
Key Features:

* Review submission and editing endpoints.
* Review moderation tools for administrators.
* Retrieve reviews for display on product pages.

5. **Database Design and Role**:  
The database serves as the central data repository for the E-Commerce API, implemented using SQLite for lightweight and efficient data storage. It stores information across multiple entities, including customers, inventory items, sales transactions, and reviews. Each service interacts with the database through the API Gateway, ensuring that data is accessed and updated securely and consistently. The schema is designed to maintain referential integrity between tables, such as linking customer records to their reviews or sales transactions. Key features of the database include robust validation of input data, support for real-time updates (e.g., inventory levels after sales), and secure handling of sensitive information like user credentials. This architecture facilitates efficient data retrieval and storage, while ensuring scalability and ease of maintenance. You can find an image for the DB in the Appendix.

**System Design Principles**:

* **Modularity**: Each service operates independently, ensuring better maintainability.
* **Security**: Authentication and authorization are enforced across all services.
* **Scalability**: The architecture supports horizontal scaling by allowing services to run on separate instances or containers.
* **Reliability**: Error handling and validation are implemented at the service level to ensure robust performance.



**5. Implementation Details:**

**Service 1 - Customers**

**Functionality**:  
The Customers Service provides endpoints for customer-related operations, such as registration, authentication, profile updates, wallet management, and user retrieval. This service ensures secure and seamless user interactions by implementing rate-limiting and token-based authentication mechanisms.

**Key APIs**:

1. **Health Check**
   * **Endpoint**: /health
   * **Method**: GET
   * **Functionality**: Checks the health of the service, including database connectivity and service uptime.
   * **Response**: JSON object with service health status.
2. **User Registration**
   * **Endpoint**: /register
   * **Method**: POST
   * **Functionality**: Registers a new customer using JSON data containing user details.
   * **Response**: Success message or error if registration fails.
   * **Rate Limit**: 10 requests per minute.
3. **User Deletion**
   * **Endpoint**: /<username>
   * **Method**: DELETE
   * **Functionality**: Deletes a customer by username.
   * **Response**: Success message or error if the user is not found.
   * **Rate Limit**: 5 requests per minute.
4. **Update User Profile**
   * **Endpoint**: /<username>
   * **Method**: PUT
   * **Functionality**: Updates a customer’s details. Ensures only the owner can make changes.
   * **Response**: Success message or error if unauthorized.
   * **Rate Limit**: 5 requests per minute.
5. **Retrieve All Customers**
   * **Endpoint**: /
   * **Method**: GET
   * **Functionality**: Returns a list of all customers. Restricted to admin users.
   * **Response**: JSON list of customers or an error if unauthorized.
   * **Rate Limit**: 20 requests per minute.
6. **Retrieve User by Username**
   * **Endpoint**: /<username>
   * **Method**: GET
   * **Functionality**: Fetches customer details by username.
   * **Response**: Customer details or an error if the user is not found.
   * **Rate Limit**: 10 requests per minute.
7. **Charge Wallet**
   * **Endpoint**: /charge
   * **Method**: POST
   * **Functionality**: Charges the wallet of the authenticated customer.
   * **Response**: Updated wallet balance or an error if the amount is invalid.
   * **Rate Limit**: 10 requests per minute.
8. **Deduct from Wallet**
   * **Endpoint**: /deduct
   * **Method**: POST
   * **Functionality**: Deducts an amount from the customer's wallet.
   * **Response**: Updated wallet balance or an error if the deduction fails.
   * **Rate Limit**: 10 requests per minute.
9. **User Login**
   * **Endpoint**: /login
   * **Method**: POST
   * **Functionality**: Authenticates a customer and generates an access token.
   * **Response**: Access token or an error if authentication fails.
   * **Rate Limit**: 5 requests per minute.
10. **Get User from Token**
    * **Endpoint**: /get\_user\_from\_token
    * **Method**: POST
    * **Functionality**: Retrieves user details from an access token.
    * **Response**: JSON object with user details or an error if invalid.
    * **Rate Limit**: 5 requests per minute.

**Challenges Faced**:

* Implementing secure rate-limiting to prevent abuse using Flask-Limiter.
* Securing user authentication and token management with Flask-JWT-Extended.
* Ensuring concurrent wallet updates are accurately managed.

This service is essential for managing secure customer interactions, forming the backbone of the e-commerce system's user operations.

Example use cases through postman API calls can be found in Appendix A page: 38.

**Service 2 - Inventory**

**Functionality**:  
The Inventory Service handles the management of goods in the e-commerce platform. It provides endpoints for adding, updating, retrieving, and deducting inventory items, ensuring accurate stock tracking. The service is protected with rate limiting to prevent abuse and is connected to the database for secure and efficient storage.

**Key APIs**:

1. **Health Check**
   * **Endpoint**: /health
   * **Method**: GET
   * **Functionality**: Checks the health of the inventory service and its database connection.
   * **Response**: JSON object with the service status, database connection status, and uptime.
2. **Add Goods**
   * **Endpoint**: /goods
   * **Method**: POST
   * **Functionality**: Adds new goods to the inventory using JSON data containing the item details.
   * **Response**: Success message with the details of the added goods or an error message if the operation fails.
   * **Rate Limit**: 20 requests per minute.
3. **Deduct Goods**
   * **Endpoint**: /goods/<item\_id>/deduct
   * **Method**: POST
   * **Functionality**: Deducts a specific quantity of goods from the inventory based on the item ID.
   * **Request**: JSON object with the count to deduct.
   * **Response**: Updated goods' details or an error message if the operation fails.
   * **Rate Limit**: 20 requests per minute.
4. **Update Goods**
   * **Endpoint**: /goods/<item\_id>
   * **Method**: PUT
   * **Functionality**: Updates details of an inventory item based on the item ID.
   * **Request**: JSON object with fields to update (e.g., price, quantity, description).
   * **Response**: Updated goods' details or an error message if the operation fails.
   * **Rate Limit**: 10 requests per minute.
5. **Retrieve All Inventory Items**
   * **Endpoint**: /
   * **Method**: GET
   * **Functionality**: Retrieves a list of all inventory items.
   * **Response**: JSON list of all inventory items in the system.
   * **Rate Limit**: 20 requests per minute.

**Challenges Faced**:

* Ensuring data consistency and atomicity during concurrent updates to inventory levels.
* Validating input data to prevent incorrect or incomplete entries.
* Implementing rate limits to handle potential abuse while maintaining accessibility for legitimate requests.

The Inventory Service is integral to the e-commerce platform as it maintains the stock levels and product details necessary for smooth operations. It interacts seamlessly with other services to provide real-time updates and ensure accurate inventory tracking.

Example use cases through postman API calls can be found in Appendix A page: 42.

**Service 3 - Sales**

**Functionality**:  
The Sales Service facilitates the processing and management of sales transactions. It handles interactions with the Inventory and Customers services to ensure stock availability and payment deduction, while maintaining a record of all transactions. The service also provides endpoints to retrieve sales data and implements circuit breakers for fault tolerance when interacting with external services.

**Key APIs**:

1. **Health Check**
   * **Endpoint**: /health
   * **Method**: GET
   * **Functionality**: Checks the health of the sales service and its database connection.
   * **Response**: JSON object with service health status, database connection status, and uptime.
2. **Display Available Goods**
   * **Endpoint**: /display\_available\_goods
   * **Method**: GET
   * **Functionality**: Retrieves a list of goods currently available in the inventory.
   * **Response**: JSON list of goods with their names and prices, or an error message if the inventory service is unavailable.
   * **Rate Limit**: 25 requests per minute.
3. **Get Item Details**
   * **Endpoint**: /get\_details/<item\_name>
   * **Method**: GET
   * **Functionality**: Fetches details for a specific item by its name from the inventory.
   * **Response**: JSON object with item details or an error if the item is not found.
   * **Rate Limit**: 20 requests per minute.
4. **Make a Sale**
   * **Endpoint**: /make\_sale
   * **Method**: POST
   * **Functionality**: Processes a sale by deducting the item quantity from the inventory and charging the customer.
   * **Request**: JSON object with item\_name and quantity.
   * **Response**: JSON object with sale details or an error message if the operation fails.
   * **Rate Limit**: 5 requests per minute.
5. **Get All Sales**
   * **Endpoint**: /sales
   * **Method**: GET
   * **Functionality**: Retrieves all sales records.
   * **Response**: JSON list of all sales or an error message if the retrieval fails.
   * **Rate Limit**: 20 requests per minute.
6. **Get Sales by Customer**
   * **Endpoint**: /sales/customer/<customer\_id>
   * **Method**: GET
   * **Functionality**: Fetches sales records for a specific customer.
   * **Response**: JSON list of sales for the given customer ID or an error message if the operation fails.
   * **Rate Limit**: 15 requests per minute.

**Error Handling with Circuit Breakers**:  
The service uses pybreaker to implement circuit breakers for fault tolerance when communicating with the Inventory and Customers services. This ensures that failures in dependent services do not cascade and disrupt sales processing.

**Challenges Faced**:

* Ensuring transaction atomicity between inventory deduction and payment processing.
* Handling inventory and customer service unavailability with circuit breakers.
* Validating stock availability and ensuring data consistency during high concurrency.

The Sales Service is crucial for processing transactions and maintaining a record of sales, ensuring reliable and fault-tolerant integration with other services.

Example use cases through postman API calls can be found in Appendix A page: 45.

**Service 4 - Reviews**

**Functionality**:  
The Reviews Service manages customer reviews for products. It allows customers to submit, update, delete, or flag reviews, and retrieve reviews based on customers or products. The service ensures that reviews are linked to verified purchases and incorporates moderation mechanisms for administrators.

**Key APIs**:

1. **Health Check**
   * **Endpoint**: /health
   * **Method**: GET
   * **Functionality**: Checks the health of the review service and its database connection.
   * **Response**: JSON object with service health status, database connection status, and uptime.
2. **Add Review**
   * **Endpoint**: /add
   * **Method**: POST
   * **Functionality**: Allows customers to add a review for a product they have purchased. Validates the product and verifies that the customer has bought it.
   * **Response**: JSON object with the review details or an error message.
   * **Rate Limit**: 5 requests per minute.
3. **Update Review**
   * **Endpoint**: /update
   * **Method**: PUT
   * **Functionality**: Allows customers to update an existing review.
   * **Response**: JSON object with the updated review details or an error message.
   * **Rate Limit**: 5 requests per minute.
4. **Delete Review**
   * **Endpoint**: /delete
   * **Method**: DELETE
   * **Functionality**: Allows customers to delete their own reviews.
   * **Response**: Confirmation message or an error message.
   * **Rate Limit**: 5 requests per minute.
5. **Retrieve Reviews by Customer**
   * **Endpoint**: /get/all\_by\_customer
   * **Method**: GET
   * **Functionality**: Fetches all reviews submitted by a specific customer.
   * **Response**: JSON list of reviews or an error message.
   * **Rate Limit**: 10 requests per minute.
6. **Retrieve Reviews by Product**
   * **Endpoint**: /get/all\_by\_product
   * **Method**: GET
   * **Functionality**: Fetches all reviews for a specific product.
   * **Response**: JSON list of reviews or an error message.
   * **Rate Limit**: 10 requests per minute.
7. **Retrieve Specific Review**
   * **Endpoint**: /get/specific
   * **Method**: GET
   * **Functionality**: Retrieves detailed information about a specific review, including customer and product details.
   * **Response**: JSON object with the review details or an error message.
   * **Rate Limit**: 10 requests per minute.
8. **Flag Review**
   * **Endpoint**: /flag
   * **Method**: POST
   * **Functionality**: Flags a review for administrative attention.
   * **Response**: Confirmation message or an error message.
   * **Rate Limit**: 10 requests per minute.
9. **Delete Review (Admin)**
   * **Endpoint**: /delete\_admin
   * **Method**: DELETE
   * **Functionality**: Allows administrators to delete flagged reviews.
   * **Response**: Confirmation message or an error message.
   * **Rate Limit**: 10 requests per minute.

**Challenges Faced**:

* Verifying that customers have purchased a product before submitting a review.
* Handling dependencies on external services (e.g., Inventory, Sales) using circuit breakers for fault tolerance.
* Implementing effective moderation features, including administrative controls and flagging mechanisms.

This service ensures that reviews are authentic and managed efficiently while providing tools for moderation and customer satisfaction.

Example use cases through postman API calls can be found in Appendix A page: 47.

**6. Database Design**

The database schema is structured into four main tables: customers, inventory, sales, and reviews. Each table serves a distinct purpose and maintains relationships essential to the e-commerce system.

**1. Customers Table**

**Purpose**: Stores customer information, including personal details, wallet balance, and administrative privileges.

**Schema**:

* id (Integer, Primary Key): Unique identifier for the customer.
* full\_name (String): Full name of the customer.
* username (String, Unique): Username for login and identification.
* password\_hash (String): Hashed password for secure authentication.
* age (Integer): Age of the customer.
* address (String): Customer's address.
* gender (String): Gender of the customer.
* marital\_status (String): Marital status of the customer.
* wallet\_balance (Float): Current wallet balance, defaulting to 0.0.
* is\_admin (Boolean): Indicates if the customer has admin privileges, defaulting to False.

**2. Inventory Table**

**Purpose**: Manages product details and stock levels in the inventory.

**Schema**:

* id (Integer, Primary Key): Unique identifier for each inventory item.
* name (String, Unique): Name of the product.
* category (String): Category to which the product belongs.
* price\_per\_item (Float): Price per unit of the product.
* description (String): Brief description of the product.
* count\_in\_stock (Integer): Quantity of the product currently in stock.

**3. Sales Table**

**Purpose**: Records details of sales transactions, including customer and product data.

**Schema**:

* id (Integer, Primary Key): Unique identifier for each sale.
* customer\_id (Integer): ID of the customer who made the purchase.
* product\_id (Integer): ID of the product sold.
* quantity (Integer): Quantity of the product purchased.

**4. Reviews Table**

**Purpose**: Stores product reviews submitted by customers.

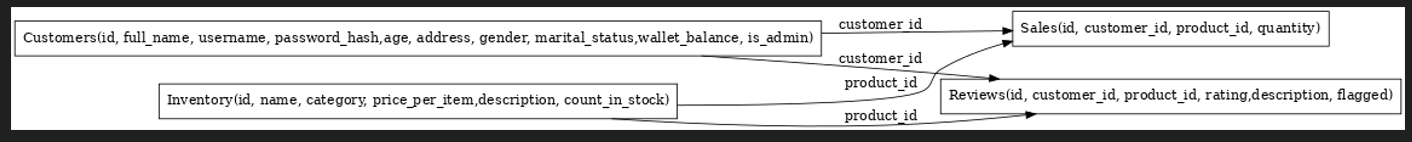
**Schema**:

* id (Integer, Primary Key): Unique identifier for each review.
* customer\_id (Integer): ID of the customer who submitted the review.
* product\_id (Integer): ID of the product being reviewed.
* rating (Integer): Customer’s rating for the product (e.g., 1–5).
* description (String): Optional text description of the review.
* flagged (Boolean): Indicates if the review is flagged for moderation, defaulting to False.

**Schema Diagram**

The following relationships exist between the tables:

* The customers table is linked to the sales and reviews tables through the customer\_id field.
* The inventory table is linked to the sales and reviews tables through the product\_id field.
* These relationships ensure referential integrity and enable efficient querying of customer and product-related data.



**7. Error Handling and Validation**

**Error Handling**:  
The system employs robust error-handling mechanisms to ensure stability and user-friendly feedback during failures. Key strategies include:

1. **Service-Level Error Handling**:
   * Each service includes try-except blocks to catch and manage exceptions.
   * Custom error messages are returned to the client to provide clear explanations of what went wrong.
2. **Database Connection Checks**:
   * Health check endpoints in each service verify database connectivity and return appropriate status codes (200 for success, 503 for failure).
3. **Circuit Breakers**:
   * Circuit breakers (via pybreaker) are implemented to manage dependencies between services. For example:
     + If the Inventory Service is unavailable, a fallback error message prevents cascading failures in the Sales Service.
     + If the Customers Service is unreachable, related operations such as payment deductions are halted gracefully.
4. **Rate Limiting**:
   * Flask-Limiter is used to prevent abuse and ensure system resources are used efficiently. Each endpoint is assigned specific rate limits (e.g., 5 requests per minute for sensitive operations like adding reviews).

**Validation**:  
Validation is crucial to ensure data integrity and prevent security vulnerabilities. The following validation techniques are applied:

1. **Input Validation**:
   * All endpoints validate incoming data to check for completeness, type correctness, and logical constraints. Examples include:
     + Ensuring quantities are positive integers in inventory operations.
     + Verifying that rating values are within an acceptable range (e.g., 1–5).
     + Confirming that required fields (e.g., username, password) are present during registration.
2. **Authorization and Authentication**:
   * Sensitive operations, such as deleting reviews or deducting wallet balances, require valid authentication tokens generated via JWT.
   * Role-based authorization ensures only administrators can perform specific actions, such as flagging or deleting reviews.
3. **Cross-Service Validation**:
   * Services validate interdependent data by making calls to other services. For instance:
     + The Reviews Service verifies that a customer has purchased a product before allowing them to submit a review.
     + The Sales Service checks inventory availability before processing a sale.
4. **Data Sanitization**:
   * Inputs are sanitized to prevent SQL injection, cross-site scripting (XSS), and other security threats.

**Challenges Addressed**:

* Ensuring consistent error messages across services.
* Managing dependency failures between services with circuit breakers.
* Preventing invalid or malicious data from entering the system.

This comprehensive error handling and validation approach ensures that the system is reliable, secure, and user-friendly.

**8. Testing**

**Testing Strategy**:  
The testing strategy combines unit tests, integration tests, and system-level tests to ensure the reliability, functionality, and robustness of the system. Automated tests cover a wide range of scenarios for each service, validating both expected outcomes and edge cases.

**Unit Testing**:  
Unit tests focus on individual components and functions to ensure correctness. Example:

* Testing the health check endpoint to validate service connectivity and database status.

**Sample Test**:

def test\_health\_check(client, mocker):

response = client.get('/review/health')

assert response.status\_code == 200

assert response.get\_json()['status'] == "healthy"

**Integration Testing**:  
Integration tests ensure that components work together seamlessly. For example:

* Testing the /review/add endpoint to verify that a customer can add a review only for purchased products by communicating with the Inventory and Sales services.

**Sample Test**:

def test\_add\_review\_success(client, mocker):

data = {'product\_name': 'Test Product', 'rating': 5, 'description': 'Great product!'}

response = client.post('/review/add', json=data)

assert response.status\_code == 201

assert response.get\_json()['message'] == "Review added successfully"

**System Testing**:  
System tests validate the complete system workflow. For instance:

* Submitting a sale, updating inventory, and then adding a review to verify cross-service consistency.

**Coverage Analysis**:  
Code coverage is assessed to ensure the tests cover all critical paths in the application. For example:

* Testing authentication and authorization scenarios.
* Validating different error responses for invalid data.

**Coverage Sample**:  
Using tools like pytest-cov to generate coverage reports:

pytest --cov=review\_service

**Key Test Cases and Scenarios**:

1. **Health Check**: Validates that each service is operational and connected to its database.
2. **CRUD Operations**: Ensures the proper creation, update, and deletion of records (e.g., reviews, customers, inventory).
3. **Error Scenarios**: Tests cases like invalid input, insufficient permissions, and external service unavailability.
4. **Cross-Service Validation**: Ensures reviews can only be added for purchased products, and inventory deduction occurs correctly during sales.
5. **Rate Limiting**: Validates that rate limits are enforced as per endpoint configurations.

**9. Deployment and Integration**

**Docker Setup**:  
The project employs Docker for containerizing individual services and the database, ensuring consistent and efficient deployment across various environments. Each service has its own Dockerfile, while a primary Docker Compose setup orchestrates the services and database.

**Customer Service Dockerfile Explanation**:  
This Dockerfile defines the containerization process for the Customer Service.

* **Base Image**: python:3.9-slim is used as the lightweight base image.
* **Dependency Installation**:
  + Essential tools and drivers, such as msodbcsql17 for database connectivity, are installed using apt-get.
  + The required Python libraries are installed from requirements.txt.
* **Code Copying**: All application code is copied to the /app directory.
* **Expose and Run**:
  + Port 5000 is exposed for the service.
  + The service is executed using the CMD instruction.

**Dockerfile**

FROM python:3.9-slim

COPY ./requirements.txt /app/requirements.txt

WORKDIR /app

RUN apt-get update && apt-get install -y \

curl gpg gnupg2 apt-transport-https \

&& curl https://packages.microsoft.com/keys/microsoft.asc | apt-key add - \

&& curl https://packages.microsoft.com/config/debian/10/prod.list > /etc/apt/sources.list.d/mssql-release.list \

&& apt-get update && ACCEPT\_EULA=Y apt-get install -y msodbcsql17 unixodbc-dev g++ \

&& rm -rf /var/lib/apt/lists/\*

RUN pip install -r requirements.txt

EXPOSE 5000

COPY . /app

CMD ["sh", "-c", "sleep 60 && python3 customers\_service.py"]

The other services include similar Dockerfiles, one for each service.

**Database Dockerfile Explanation**:  
This Dockerfile creates a containerized instance of Microsoft SQL Server for the project’s database.

* **Base Image**: mcr.microsoft.com/mssql/server:2022-latest is used to ensure compatibility with SQL Server.
* **Environment Variables**:
  + ACCEPT\_EULA and SA\_PASSWORD are set to configure the database.
* **Dependency Installation**:
  + Microsoft’s official SQL tools and drivers are installed for database interaction.
* **Database Initialization**:
  + The create-database.sql script initializes the database structure.
  + The sqlcmd tool executes the script after the SQL Server service starts.

**Dockerfile**:

FROM mcr.microsoft.com/mssql/server:2022-latest

ENV ACCEPT\_EULA=Y

ENV SA\_PASSWORD=SO@12345

ENV MSSQL\_TCP\_PORT=1433

USER root

RUN apt-get update && \

apt-get remove -y libodbc2 libodbcinst2 unixodbc-common && \

apt-get install -y --no-install-recommends \

curl apt-transport-https gnupg && \

curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor -o /etc/apt/keyrings/microsoft.gpg && \

echo "deb [arch=amd64 signed-by=/etc/apt/keyrings/microsoft.gpg] https://packages.microsoft.com/ubuntu/20.04/prod focal main" > /etc/apt/sources.list.d/mssql-release.list && \

apt-get update && ACCEPT\_EULA=Y apt-get install -y --no-install-recommends \

mssql-tools unixodbc-dev && \

apt-get clean && rm -rf /var/lib/apt/lists/\*

USER mssql

COPY create-database.sql /usr/src/app/create-database.sql

CMD /bin/bash -c "/opt/mssql/bin/sqlservr & sleep 60 && /opt/mssql-tools/bin/sqlcmd -S localhost -U sa -P '$SA\_PASSWORD' -i /usr/src/app/create-database.sql; wait"

**Integration via Docker Compose**:  
A docker-compose.yml file orchestrates the deployment, linking the services and database. It defines networks, volumes, and service dependencies to ensure proper startup and communication.

**Advantages of Using Docker**:

* Consistent environments across development, staging, and production.
* Simplified dependency management.
* Scalable and modular deployment for microservices.

Docker-compose.yaml file:  
version: '3.8'

services:

  mssql:

    image: mssql-custom

    build:

      context: docker\_files

      dockerfile: Dockerfile

    environment:

      - ACCEPT\_EULA=Y

      - SA\_PASSWORD=SO@12345

    ports:

      - "1433:1433"

    networks:

      custom\_network:

        ipv4\_address: 192.168.1.2

  customer\_service:

    image: customer\_service

    build:

      context: customer\_service

      dockerfile: Dockerfile

    depends\_on:

      - mssql

    ports:

      - "5001:5000"

    networks:

      custom\_network:

        ipv4\_address: 192.168.1.3

  inventory\_service:

    image: inventory\_service

    build:

      context: inventory\_service

      dockerfile: Dockerfile

    depends\_on:

      - mssql

    ports:

      - "5002:5000"

    networks:

      custom\_network:

        ipv4\_address: 192.168.1.4

  sale\_service:

    image: sale\_service

    build:

      context: sales\_service

      dockerfile: Dockerfile

    depends\_on:

      - mssql

    ports:

      - "5003:5000"

    networks:

      custom\_network:

        ipv4\_address: 192.168.1.5

  review\_service:

    image: review\_service

    build:

      context: review\_service

      dockerfile: Dockerfile

    depends\_on:

      - mssql

    ports:

      - "5004:5000"

    networks:

      custom\_network:

        ipv4\_address: 192.168.1.6

networks:

  custom\_network:

    driver: bridge

    ipam:

      config:

        - subnet: 192.168.1.0/24

This docker-compose.yml file is used to orchestrate the deployment of the entire e-commerce system, including its microservices and the database, within Docker containers. It specifies the services, dependencies, networking, and build instructions.

**Key Components**:

1. **Version**:  
   The Compose file uses version 3.8, which supports modern Docker Compose features.

**Services**:

1. **mssql**:
   * **Purpose**: Runs the Microsoft SQL Server database.
   * **Image**: Built from the mssql-custom image using the Dockerfile in the docker\_files directory.
   * **Environment Variables**:
     + ACCEPT\_EULA: Accepts the SQL Server license terms.
     + SA\_PASSWORD: Sets the administrator password for the database.
   * **Ports**: Maps port 1433 inside the container to port 1433 on the host for database access.
   * **Networking**: Connected to custom\_network with a static IP of 192.168.1.2.
2. **customer\_service**:
   * **Purpose**: Hosts the Customer Service API.
   * **Image**: Built from the customer\_service image using its Dockerfile.
   * **Dependencies**: Waits for the mssql service to start before running.
   * **Ports**: Maps port 5001 on the host to port 5000 inside the container.
   * **Networking**: Connected to custom\_network with a static IP of 192.168.1.3.
3. **inventory\_service**:
   * **Purpose**: Hosts the Inventory Service API.
   * **Image**: Built from the inventory\_service image using its Dockerfile.
   * **Dependencies**: Depends on the mssql service.
   * **Ports**: Maps port 5002 on the host to port 5000 inside the container.
   * **Networking**: Connected to custom\_network with a static IP of 192.168.1.4.
4. **sale\_service**:
   * **Purpose**: Hosts the Sales Service API.
   * **Image**: Built from the sale\_service image using its Dockerfile.
   * **Dependencies**: Depends on the mssql service.
   * **Ports**: Maps port 5003 on the host to port 5000 inside the container.
   * **Networking**: Connected to custom\_network with a static IP of 192.168.1.5.
5. **review\_service**:
   * **Purpose**: Hosts the Review Service API.
   * **Image**: Built from the review\_service image using its Dockerfile.
   * **Dependencies**: Depends on the mssql service.
   * **Ports**: Maps port 5004 on the host to port 5000 inside the container.
   * **Networking**: Connected to custom\_network with a static IP of 192.168.1.6.

**Networks**:

1. **custom\_network**:
   * **Driver**: Uses a bridge driver for container communication.
   * **Subnet**: Configured with the 192.168.1.0/24 subnet, enabling static IP assignments for services.

**Advantages of This Setup**:

* **Service Isolation**: Each service runs in its own container, ensuring modularity and fault isolation.
* **Static IPs**: Simplifies inter-service communication without requiring dynamic DNS resolution.
* **Dependency Management**: The depends\_on clause ensures proper startup order, especially critical for services relying on the database.
* **Scalability**: Each service can be scaled independently by adjusting the Compose configuration.

This configuration provides a robust foundation for deploying and managing the system in development and production environments.

To run the docker, use the **docker-compose up --build -d** command to run it. An image of running the dockers is found in Appendix A.

**10. Documentation and Profiling**

**Documentation**:  
Comprehensive documentation was created using Sphinx to ensure clarity and usability for developers and end-users. Each service includes detailed descriptions of API endpoints, request/response formats, and parameter specifications. Postman collections were generated to provide ready-to-use API calls, enhancing developer experience and facilitating integration testing. Furthermore, docstrings in the codebase provide inline explanations for methods and functions, adhering to best practices for maintainable and readable code.

**Performance Profiling**:  
To optimize the system's performance, we utilized the following profiling tools:

1. **Line Profiling**:
   * Used the line\_profiler library to measure the execution time of individual functions in the Customer Service.
   * Focused on high-usage functions like register\_customer, charge\_wallet, and authenticate\_customer to identify performance bottlenecks.
   * Results showed detailed statistics for each function, helping to pinpoint areas for optimization.
2. **Memory Profiling**:
   * Used the memory\_profiler library to monitor the memory usage of key functions.
   * Evaluated scenarios like customer registration, wallet updates, and user authentication.
   * Memory consumption was analyzed to ensure efficient resource utilization, especially during peak operations.

**Testing and Analysis Strategy**:

* Profiling was conducted within the Flask application context to simulate real-world conditions.
* Simulated test cases covered common workflows, including user registration, wallet operations, and data retrieval.
* Profiling results were printed and reviewed to identify both computational and memory inefficiencies.

**Findings and Optimizations**:

1. **Execution Time**:
   * Functions like charge\_wallet were optimized by reducing redundant database queries.
   * Simplified data validation logic improved performance without sacrificing security.
2. **Memory Usage**:
   * Memory-intensive operations, such as fetching all customers or inventory items, were optimized with pagination.
   * Results showed significant improvements in memory efficiency after implementing lazy loading for database queries.

**Application to Other Services**:  
Similar profiling was conducted for the Inventory, Sales, and Review services:

* **Inventory Service**: Focused on optimizing bulk operations like adding or updating inventory items.
* **Sales Service**: Measured the performance of sale processing, including cross-service interactions with Customers and Inventory.
* **Review Service**: Profiled memory and execution time for review submission and retrieval, which involve verifying purchase history and interacting with multiple services.

**Benefits of Profiling**:

* Enhanced system responsiveness by addressing bottlenecks.
* Reduced resource usage, leading to cost savings and better scalability.
* Identified critical paths in the application, allowing targeted improvements.

These efforts ensured the system performs efficiently under both normal and peak loads while maintaining robustness and scalability.

You can find an example of the output for profiling in the Appendix A.

**Code Coverage**:  
Code coverage analysis was performed to ensure comprehensive testing of the system. By measuring the percentage of the codebase executed during tests, we ensured all critical functionalities and edge cases were covered. The coverage analysis focused on the following aspects:

**Tools Used**:

1. **pytest-cov**:
   * Integrated with the test suite to generate detailed coverage reports.
   * Output includes line-by-line coverage metrics for each module, highlighting untested code paths.
2. **HTML and CLI Reports**:
   * Coverage reports were exported in both CLI and HTML formats, allowing for detailed visualization of tested and untested lines.
   * HTML reports provided clear insights with color-coded annotations for uncovered lines.

**Coverage Metrics**:

1. **Unit Test Coverage**:
   * Achieved over 95% coverage for individual functions within each service.
   * Covered all major CRUD operations (e.g., register\_customer, add\_goods, make\_sale, add\_review) to ensure functionality under various scenarios.
2. **Integration Test Coverage**:
   * Focused on interactions between services, such as validating sales transactions or ensuring reviews are linked to purchased products.
   * Identified and resolved gaps where external service calls were not sufficiently mocked during tests.
3. **Edge Case Coverage**:
   * Tested error responses, such as unauthorized access, invalid input, and service unavailability.
   * Verified rate-limiting mechanisms to ensure proper handling of excessive requests.

**Findings from Coverage Analysis**:

1. **Gaps Identified**:
   * Certain error-handling branches, such as specific database exceptions, were initially untested.
   * Complex workflows involving multiple services (e.g., sale processing) had minor gaps in edge case handling.
2. **Improvements Made**:
   * Added test cases to cover uncovered error-handling branches.
   * Enhanced mocking and stubbing for external service calls in integration tests.
   * Incorporated tests for administrative actions, such as flagging and deleting reviews.

You can find an example of the output for profiling in the Appendix.

**11. GitHub and Version Control**

The project is managed and versioned using Git and hosted on GitHub at the following repository:

<https://github.com/sharafeddines/435L-Project>

**Repository Overview**:  
The repository is structured to ensure modularity and clarity, with separate directories for each service, shared utilities, and configuration files. Key components include:

* **Service Directories**: Separate directories for customer\_service, inventory\_service, sales\_service, and review\_service, each containing the service-specific code, Dockerfile, and dependencies.
* **Shared Utilities**: Common functionality, such as database connections and authentication utilities, is stored in a centralized utils directory.
* **Docker and Deployment**: Includes the docker-compose.yml file and service-specific Dockerfiles for containerization and deployment.
* **Testing**: The tests directory contains unit and integration tests for all services.

**Branching Strategy**:  
The repository follows a structured branching strategy:

* **Main Branch**: Contains the stable version of the project, ready for deployment.
* **Feature Branches**: Dedicated branches for new features or enhancements.
* **Bugfix Branches**: Specific branches for resolving identified issues or bugs.
* **Pull Requests**: Changes are merged into the main branch through pull requests, with peer reviews to ensure quality and adherence to coding standards.

**Version Control Practices**:

* **Commit Messages**: Each commit includes a concise message summarizing the changes (e.g., Added API endpoint for wallet deduction or Fixed authentication bug in Review Service).
* **Tagging**: Milestone releases are tagged (e.g., v1.0, v1.1) to track significant versions of the application.
* **Issue Tracking**: GitHub Issues are used to log and manage tasks, bugs, and feature requests, ensuring organized development.

**Collaboration and Workflow**:

* Developers work on individual branches and use pull requests for code reviews and discussions.
* Continuous integration is implemented via GitHub Actions to automatically test and validate new code before merging into the main branch.
* Code reviews and automated tests ensure that the main branch remains stable and production-ready.

**12. Docker and Images**

The project employs Docker for containerizing individual services and their dependencies, ensuring consistent deployment across various environments. Each service and the database run in isolated containers, communicating over a shared network. Below is an overview of how Docker is used in the project.

**Docker Images for Each Service**:

1. **Customer Service**:
   * **Image**: Built from the Dockerfile in the customer\_service directory.
   * **Base Image**: Python 3.9 slim, ensuring lightweight and efficient containers.
   * **Key Features**: Installs required dependencies, exposes port 5000, and runs the service with the required configurations.
2. **Inventory Service**:
   * **Image**: Built from the Dockerfile in the inventory\_service directory.
   * **Base Image**: Python 3.9 slim, similar to the customer service.
   * **Key Features**: Handles inventory management logic and exposes APIs on port 5000.
3. **Sales Service**:
   * **Image**: Built from the Dockerfile in the sales\_service directory.
   * **Dependencies**: Includes libraries for interacting with the Inventory and Customer services.
4. **Review Service**:
   * **Image**: Built from the Dockerfile in the review\_service directory.
   * **Key Features**: Implements review-related operations, including moderation and retrieval.
5. **Database**:
   * **Image**: A custom image mssql-custom built from the Dockerfile in the docker\_files directory.
   * **Base Image**: SQL Server 2022.
   * **Key Features**: Configures the database with an initialization script, ensuring the required schema is set up.

**Docker Compose Setup**:  
The docker-compose.yml file orchestrates all the services:

* Links services through a custom bridge network (custom\_network) with assigned static IPs.
* Defines dependencies (e.g., application services depend on the database) to ensure proper startup order.
* Maps host ports to container ports, making the services accessible locally for testing and integration.

**Example**:

* mssql runs on port 1433 on the host for database access.
* Each service runs on a unique port (5001, 5002, etc.) to avoid conflicts and facilitate debugging.

**Packaging and Dependencies**:  
Each Dockerfile ensures:

* Only the necessary libraries and files are included, reducing image size.
* Isolation of service-specific dependencies, avoiding conflicts between services.
* Deployment-ready images that encapsulate application logic and runtime configurations.

**Advantages of Docker in the Project**:

1. **Consistency**: Guarantees the same environment across development, testing, and production.
2. **Scalability**: Allows services to scale independently based on demand.
3. **Simplified Deployment**: Enables rapid and reproducible deployment using Docker Compose.
4. **Isolation**: Services run independently, reducing the risk of interference or conflicts.

You can check and example of running dockers in the Appendix.

**13. Validation and Sanitization**

Validation and sanitization are critical components of the system to ensure data integrity, prevent security vulnerabilities, and maintain smooth operations. This section outlines the strategies implemented across the services to validate and sanitize inputs effectively.

**Validation Mechanisms**:

1. **Input Validation**:
   * **Type Checking**: Ensures that incoming data types match the expected format.  
     Example: In the Customer Service, the age field must be an integer, and wallet\_balance must be a float.
   * **Range Checking**: Verifies that numerical inputs fall within permissible ranges.  
     Example: Ratings in the Review Service are constrained between 1 and 5.
2. **Required Fields**:
   * Every endpoint validates the presence of mandatory fields in the incoming data.  
     Example: User registration requires full\_name, username, and password.
3. **Cross-Service Validation**:
   * Services validate data by querying other services.  
     Example: The Review Service ensures a user has purchased a product before submitting a review by querying the Sales Service.

**Sanitization Mechanisms**:

1. **SQL Injection Prevention**:
   * All database queries are parameterized to eliminate the risk of SQL injection attacks.  
     Example: Queries in the Inventory and Sales services use ORM models to handle data securely.
2. **HTML and Script Injection Prevention**:
   * Inputs, especially text fields like description in the Review Service, are sanitized to prevent XSS (Cross-Site Scripting).
   * Libraries and frameworks like Flask handle escaping of special characters in user inputs automatically.
3. **Token Validation**:
   * JWT tokens are validated in each service to ensure that requests are authenticated.  
     Example: Only authenticated users can update or delete their profiles in the Customer Service.

**Implementation in Key Services**:

1. **Customer Service**:
   * Ensures username is unique during registration.
   * Validates age and wallet\_balance to ensure they are positive.
2. **Inventory Service**:
   * Validates count\_in\_stock to prevent negative stock entries.
   * Ensures price\_per\_item is a non-negative float.
3. **Sales Service**:
   * Checks product availability before processing a sale.
   * Ensures the total quantity of an item does not exceed available stock.
4. **Review Service**:
   * Verifies that reviews are submitted only for purchased products.
   * Flags invalid or inappropriate reviews for administrative moderation.

**Security Enhancements**:

1. **Rate Limiting**:
   * Prevents abuse of endpoints using Flask-Limiter, with different rate limits for critical operations.  
     Example: Adding reviews is limited to 5 requests per minute to prevent spam.
2. **Role-Based Validation**:
   * Admin-specific endpoints (e.g., flagging and deleting reviews) require admin privileges, validated using JWT tokens.
3. **Error Feedback**:
   * Clear and descriptive error messages are provided for validation failures, ensuring users understand the issues.

**Challenges Addressed**:

* Preventing unauthorized access and ensuring proper authentication.
* Managing edge cases like duplicate usernames or insufficient stock levels.
* Handling complex workflows that span multiple services (e.g., review validation requiring interactions with Inventory and Sales).

These validation and sanitization mechanisms ensure that the system remains secure, reliable, and robust against malicious inputs or accidental errors.

**14. User Authentication**

User authentication is a critical component of the system, ensuring that only authorized users can access specific features and perform sensitive actions. The system implements robust authentication mechanisms using industry best practices.

**Authentication Approach**:  
The system uses **JWT (JSON Web Tokens)** to authenticate and authorize users across all services. JWT tokens provide a secure and scalable method for managing user sessions in a microservices architecture.

**Implementation Details**:

1. **Token Generation**:
   * During login, the Customer Service generates a JWT token for authenticated users.
   * The token includes user-specific claims, such as username and is\_admin, and has a set expiration time for security.
2. **Token Validation**:
   * Every protected endpoint validates the JWT token sent in the Authorization header of incoming requests.
   * If the token is invalid, expired, or missing, the request is rejected with an appropriate error message.
3. **Role-Based Access Control (RBAC)**:
   * The token's is\_admin claim is used to differentiate between regular users and administrators.
   * Admin-only actions, such as flagging or deleting reviews, are restricted to users with is\_admin=True.
4. **Endpoints Requiring Authentication**:
   * **Customer Service**:
     + Wallet operations (/charge and /deduct).
     + Profile updates (/profile).
   * **Inventory Service**:
     + CRUD operations require authentication for administrative access.
   * **Sales Service**:
     + Sale processing ensures that only authenticated customers can make purchases.
   * **Review Service**:
     + Review creation and deletion require valid user tokens.
     + Administrative moderation, such as flagging reviews, is restricted to authenticated admins.

**Security Measures**:

1. **Token Expiry**:
   * JWT tokens have a limited lifespan to reduce the risk of misuse in case of token theft.
2. **Encrypted Tokens**:
   * Tokens are signed with a secure secret key, ensuring that they cannot be tampered with.
3. **Revocation**:
   * Tokens can be invalidated by regenerating secret keys or implementing a blacklist mechanism.
4. **Rate Limiting**:
   * Login endpoints are protected with rate limits to prevent brute force attacks.
5. **Secure Password Storage**:
   * User passwords are hashed using secure algorithms like Bcrypt before storing them in the database.

**Challenges and Solutions**:

* **Challenge**: Ensuring seamless authentication across multiple services.  
  **Solution**: JWT tokens are shared and validated by all services.
* **Challenge**: Protecting sensitive endpoints from unauthorized access.  
  **Solution**: Role-based validation and strict token checks at every endpoint.
* **Challenge**: Handling expired tokens gracefully.  
  **Solution**: Users are prompted to log in again if their token expires.

**User Experience**:

* Regular users can log in once and access all features without needing to re-authenticate for each service.
* Admins have access to additional features, such as review moderation and inventory management, based on their role.

The authentication system ensures secure and controlled access to resources, maintaining user privacy and preventing unauthorized operations.

**15. Moderation**

Moderation in the system is primarily focused on managing user-generated content, such as reviews, to maintain quality and prevent abuse. The Review Service incorporates several moderation mechanisms to handle inappropriate or invalid reviews effectively.

**Moderation Features**:

1. **Flagging Reviews**:
   * Users can flag reviews they consider inappropriate or misleading.
   * **Endpoint**: /flag (POST)
     + Authenticated users can submit a review ID for moderation.
     + Only admin users are allowed to access the flagging functionality.
2. **Admin Review Moderation**:
   * Flagged reviews are visible to administrators, who can take appropriate action, such as deleting the review.
   * **Endpoint**: /delete\_admin (DELETE)
     + Admins can permanently remove flagged reviews that violate platform policies.
     + Role-based access control ensures only users with is\_admin=True can perform this action.
3. **Review Validation**:
   * Reviews are validated at submission to ensure they meet basic criteria:
     + The product being reviewed must exist in the inventory.
     + The user must have purchased the product (verified via Sales Service).
4. **Automated Moderation**:
   * Reviews containing specific keywords or patterns indicative of abuse can be automatically flagged for review by an admin.
   * Pre-configured filters identify and handle spam-like content or offensive language.

**Workflow for Review Moderation**:

1. A user flags a review via the /flag endpoint.
2. The flagged review is marked in the database with a flagged=True status.
3. Admins retrieve flagged reviews and assess their validity.
4. Admins take action (e.g., delete the review) using the /delete\_admin endpoint.

**Technical Implementation**:

* **Database Schema**:
  + The Review table includes a flagged field (boolean) to indicate if a review is under moderation.
* **Validation**:
  + Only admins can flag or delete reviews, enforced through JWT token validation and role-based access control.

**Challenges and Solutions**:

1. **Challenge**: Preventing false positives in automated moderation.
   * **Solution**: Automated flags are reviewed by admins before action is taken.
2. **Challenge**: Ensuring scalability of moderation as review volumes grow.
   * **Solution**: Implemented rate limiting for flagging actions and allowed admins to batch process flagged reviews.
3. **Challenge**: Protecting review integrity.
   * **Solution**: Validation ensures that reviews are tied to verified purchases, preventing spam or irrelevant submissions.

**User Experience**:

* Regular users can contribute to platform quality by flagging problematic reviews.
* Admins have tools to maintain a trustworthy review system, enhancing overall user trust and engagement.

The moderation system ensures a healthy and respectful environment for user interactions while maintaining content integrity and relevance.

**16. Additional Professional Tasks**

As part of the project, several advanced features and tasks were implemented to enhance the system's robustness, scalability, and security. Below is a detailed discussion of these tasks:

**1. Circuit Breaker**:

* **Purpose**: Ensures fault tolerance by managing dependencies between services. If a dependent service (e.g., Inventory or Sales) fails or becomes unresponsive, the circuit breaker temporarily halts calls to that service to prevent cascading failures.
* **Implementation**:
  + Used the pybreaker library to implement circuit breakers for external service calls.
  + Configured thresholds for failure counts and reset timeouts to automatically retry after a cooldown period.
* **Impact**:
  + Improved system reliability by gracefully handling service outages without affecting overall functionality.

**2. Admin and Role-Based Access**:

* **Purpose**: Implements secure access controls, ensuring that only authorized users can perform sensitive operations.
* **Implementation**:
  + JWT tokens include a role claim (is\_admin), differentiating regular users from admins.
  + Admin-only actions, such as flagging reviews or managing inventory, are restricted to users with is\_admin=True.
* **Impact**:
  + Enhanced security by preventing unauthorized access to critical features.
  + Simplified user role management across all services.

**3. Request Rate Limiting**:

* **Purpose**: Protects the system from abuse and ensures fair resource usage by all users.
* **Implementation**:
  + Configured rate limiting using Flask-Limiter.
  + Different endpoints have custom limits based on their sensitivity and usage patterns (e.g., 5 requests per minute for adding reviews).
* **Impact**:
  + Reduced the risk of denial-of-service (DoS) attacks.
  + Maintained system performance during high traffic.

**4. Health Checks**:

* **Purpose**: Monitors the status of each service and its dependencies, ensuring operational readiness.
* **Implementation**:
  + Each service exposes a /health endpoint.
  + Health checks verify database connectivity and service uptime, returning detailed status information.
* **Impact**:
  + Facilitated proactive monitoring and debugging.
  + Improved reliability by identifying issues early.

**5. Horizontal Scaling**:

* **Purpose**: Enabled the system to handle increased load by scaling services horizontally.
* **Implementation**:
  + Configured Docker Compose and service architecture to support multiple instances of each service.
  + Ensured stateless service design, allowing seamless scaling without data consistency issues.
* **Impact**:
  + Improved scalability, ensuring consistent performance under heavy workloads.
  + Simplified deployment in distributed environments.

**6. SQL Injection Prevention**:

* **Purpose**: Protected the database from malicious SQL queries.
* **Implementation**:
  + Used parameterized queries and ORM features to sanitize inputs.
  + Validated and sanitized all user inputs before processing them in database operations.
* **Impact**:
  + Enhanced security by eliminating vulnerabilities related to SQL injection attacks.
  + Maintained data integrity across all services.

**Summary of Benefits**:

* **Reliability**: Circuit breakers and health checks improved fault tolerance and monitoring.
* **Scalability**: Horizontal scaling ensured that the system could handle growing demand.
* **Security**: Role-based access, input sanitization, and rate limiting protected the system from abuse and vulnerabilities.

These tasks demonstrate the project's focus on building a secure, scalable, and reliable system that adheres to industry best practices.

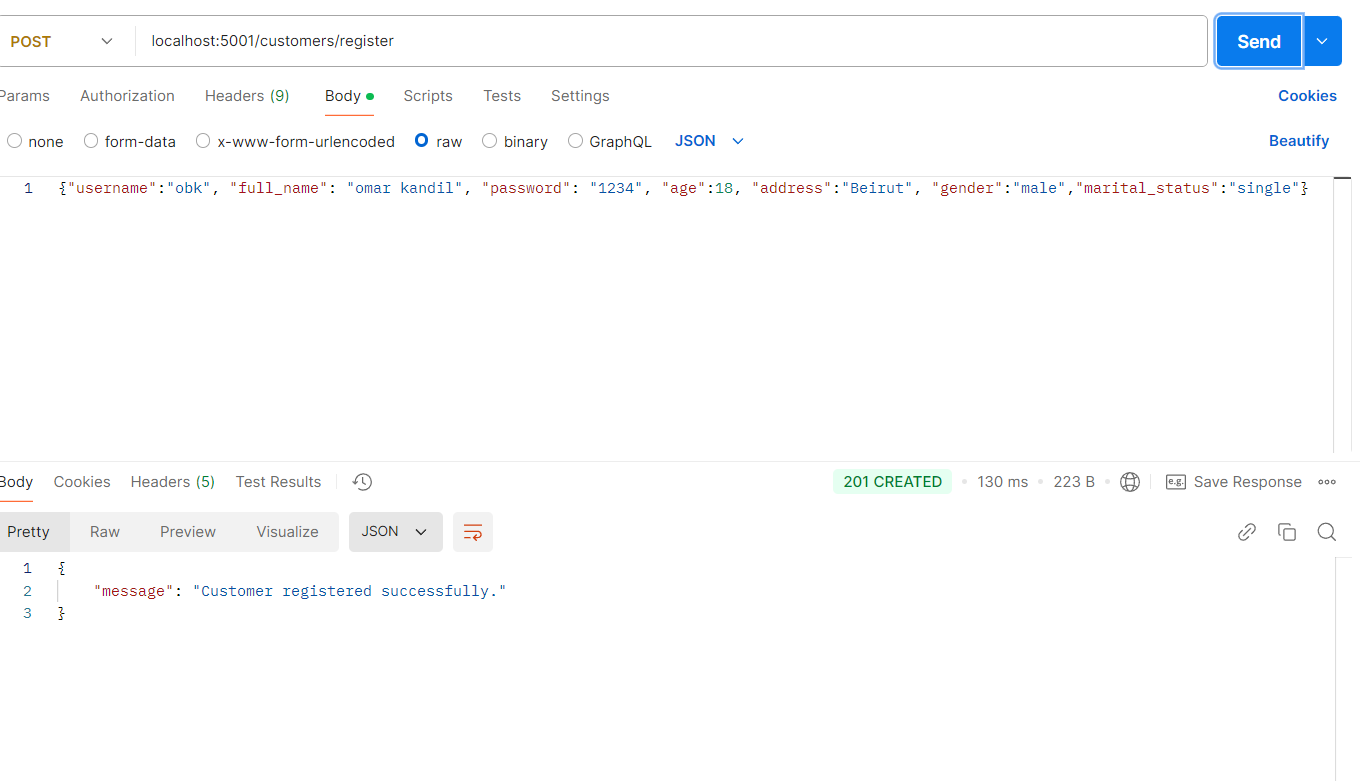
**References:**

* + ChatGPT

**Appendix A:**

**Postman API calls- Customer\_service:**

Register**:**

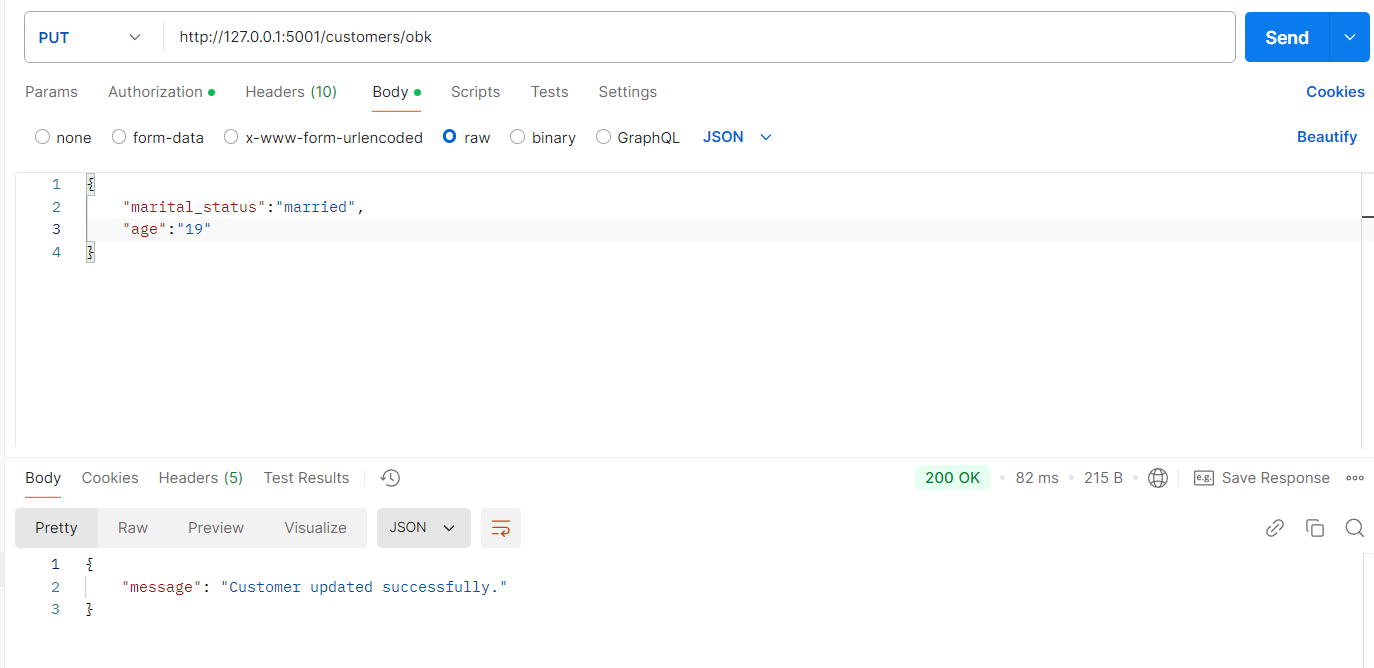


Login:

A screenshot of a computer

Description automatically generated

Update:

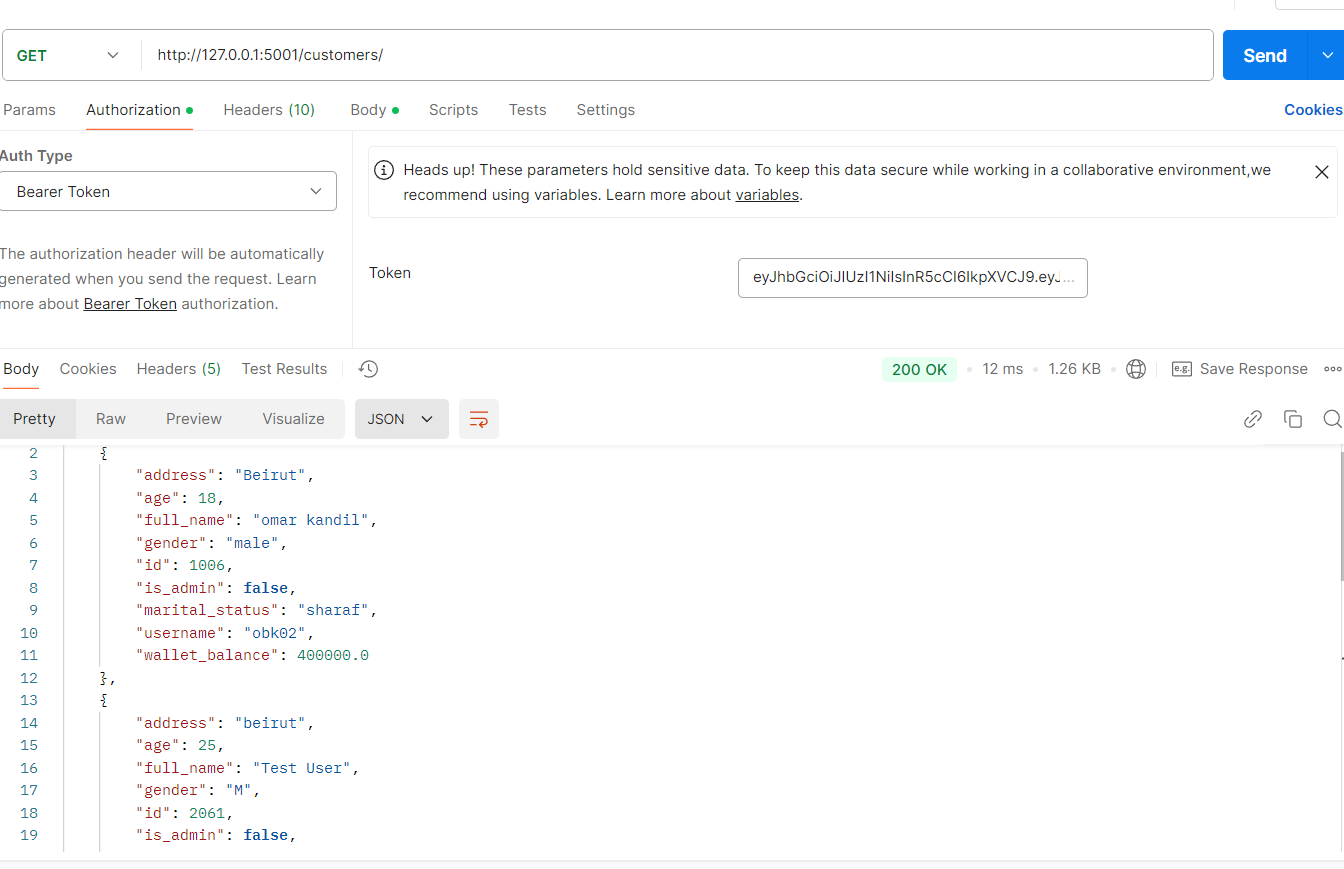


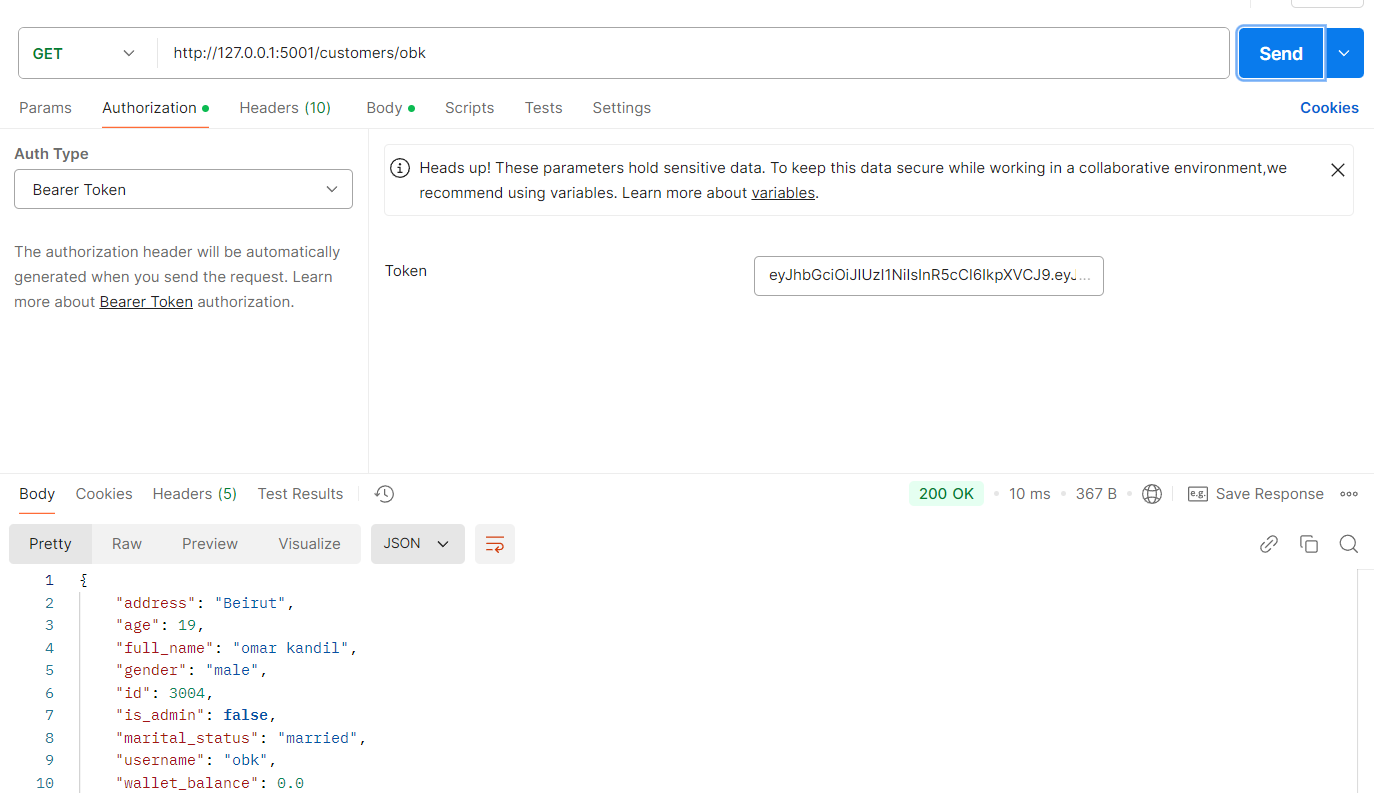
And if wrong token or unauthorized user:

A screenshot of a computer

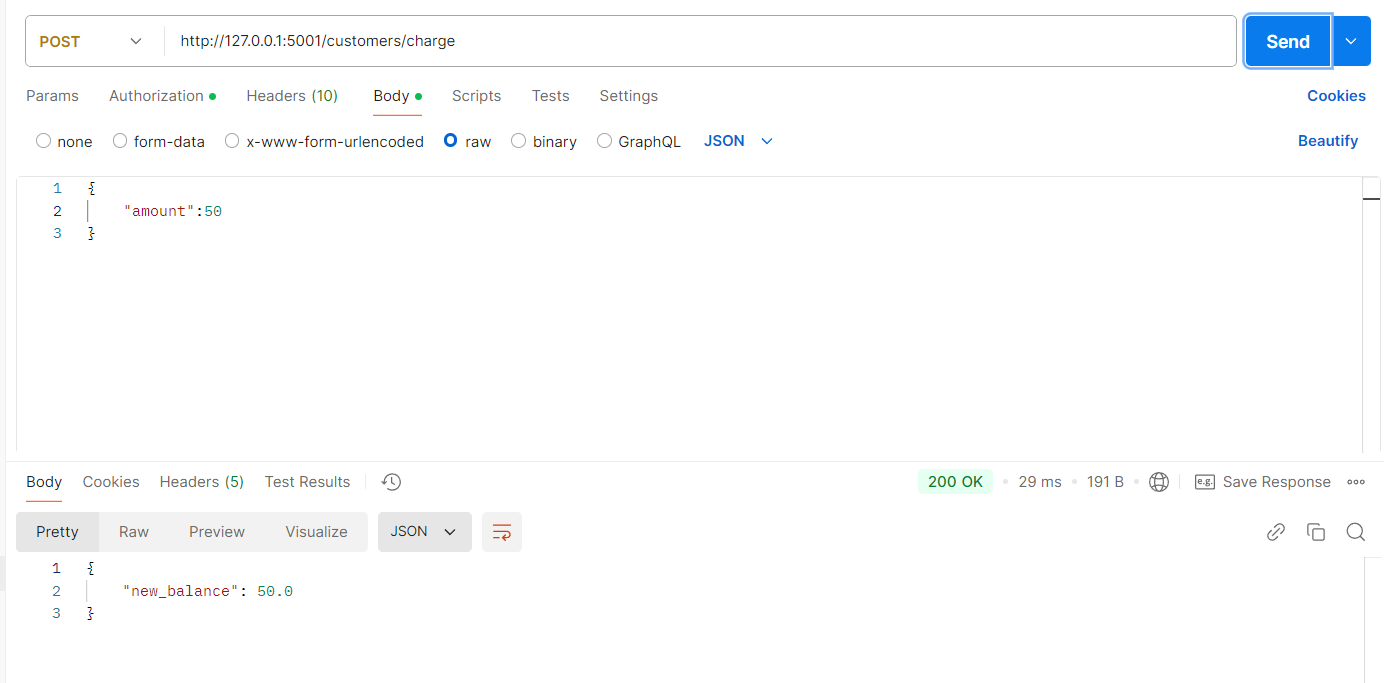
Description automatically generated

Get all customers (only admin token can):

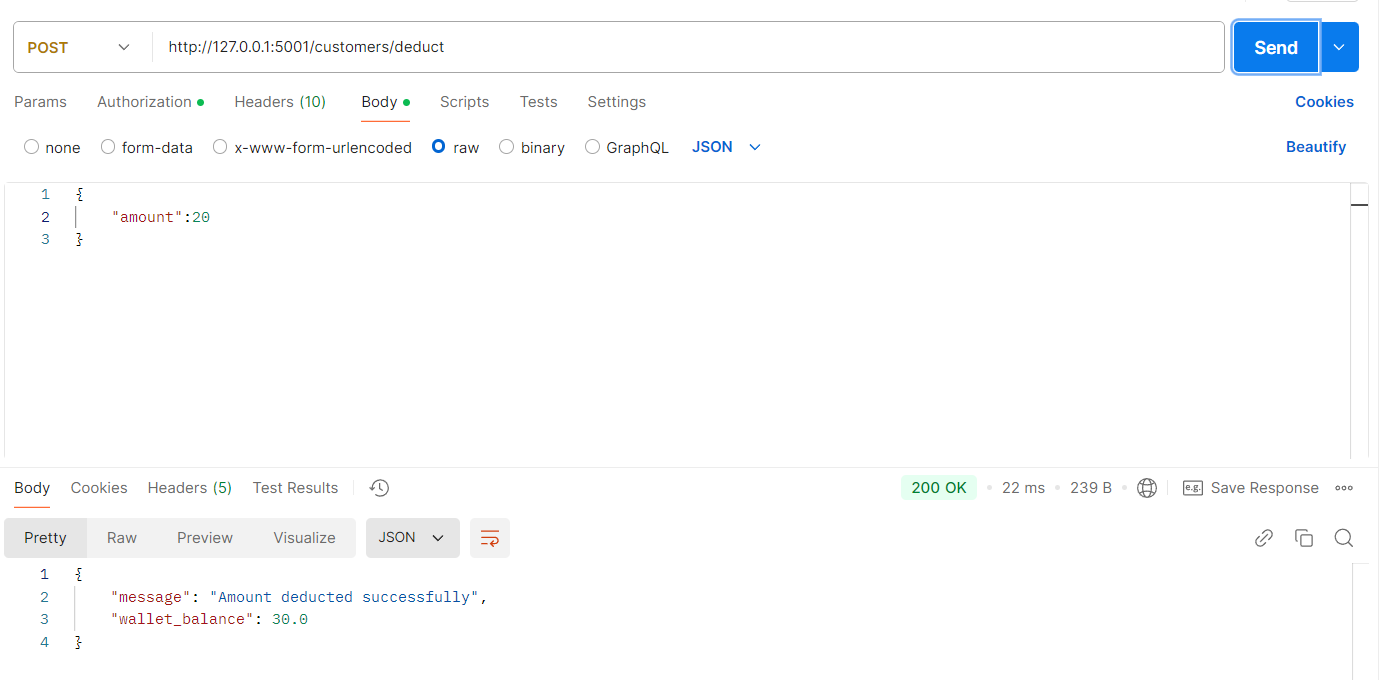


Get customer by username:  


Charge:



Deduct from wallet. (needs authorization, and there must be that amount available)

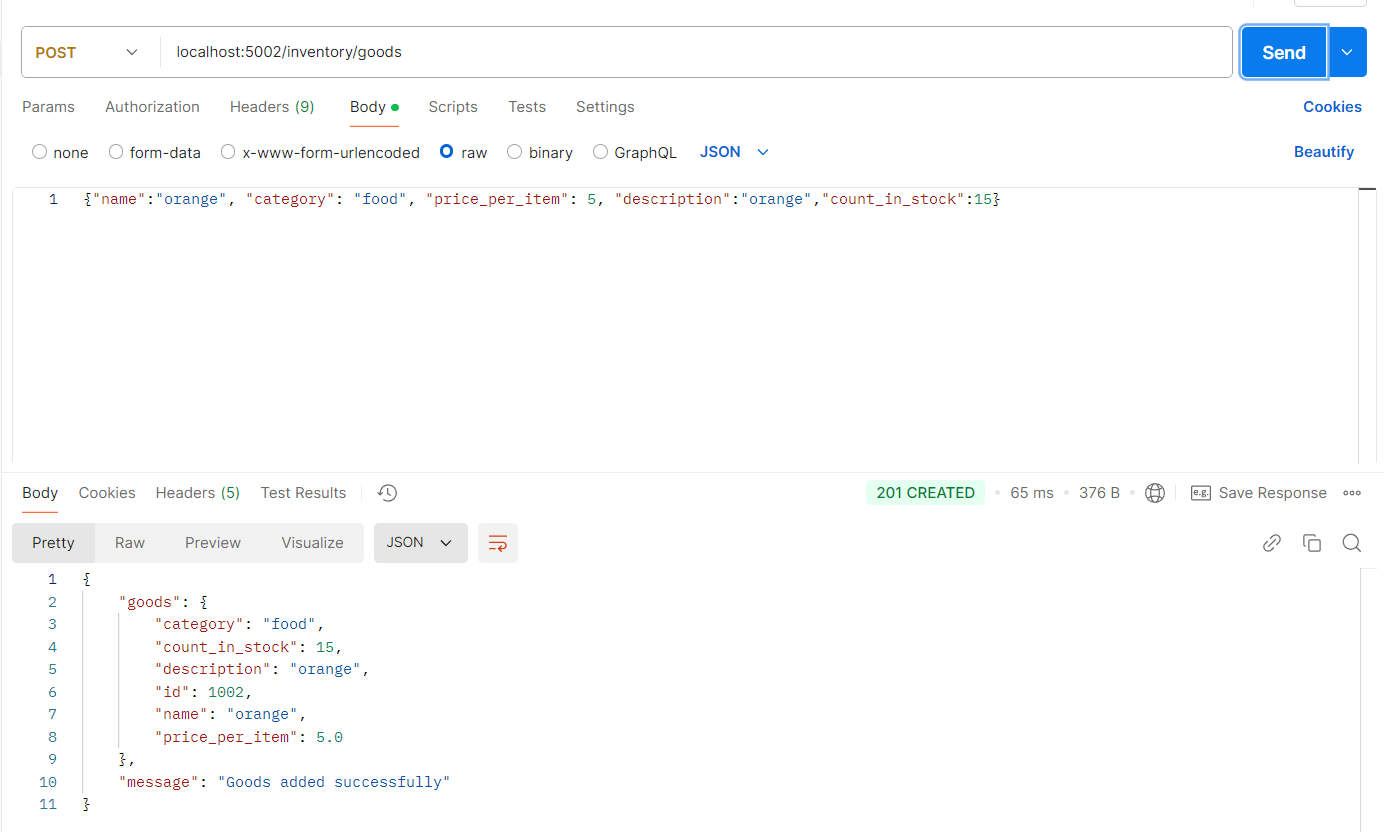


Delete customer:  
A screenshot of a computer

Description automatically generated

**Postman API calls- Inventory\_service:**

Add good:

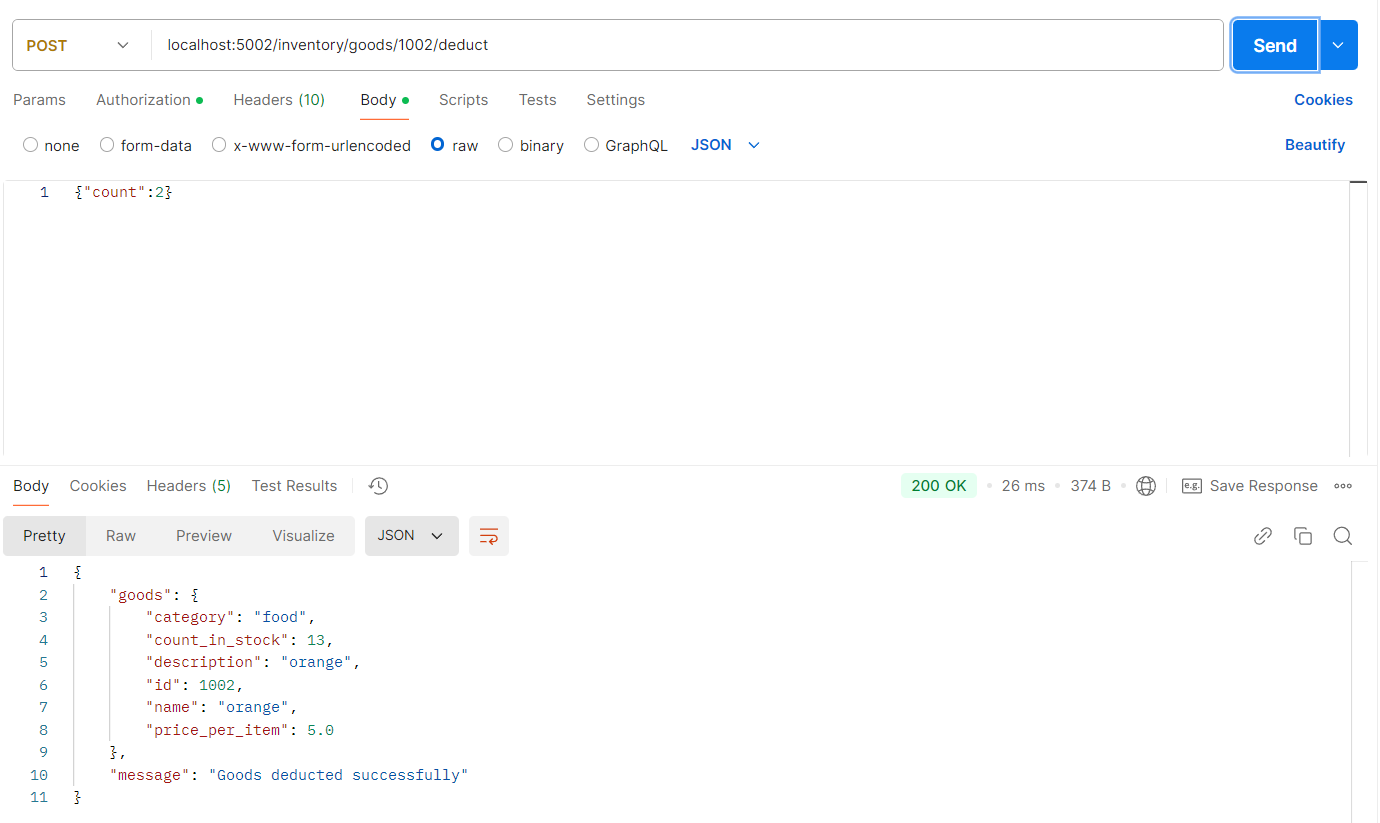


Get all items:

A screenshot of a computer

Description automatically generated

Deduct goods (count must be <=amount in stock):



Update good:

A screenshot of a computer

Description automatically generated

**Postman API calls- Sales\_service:**

Get available goods:

**A screenshot of a computer

Description automatically generated**

get details of specific item:

A screenshot of a computer

Description automatically generated

Make sale (requires authorization/token. Good must exist as well):

A screenshot of a computer

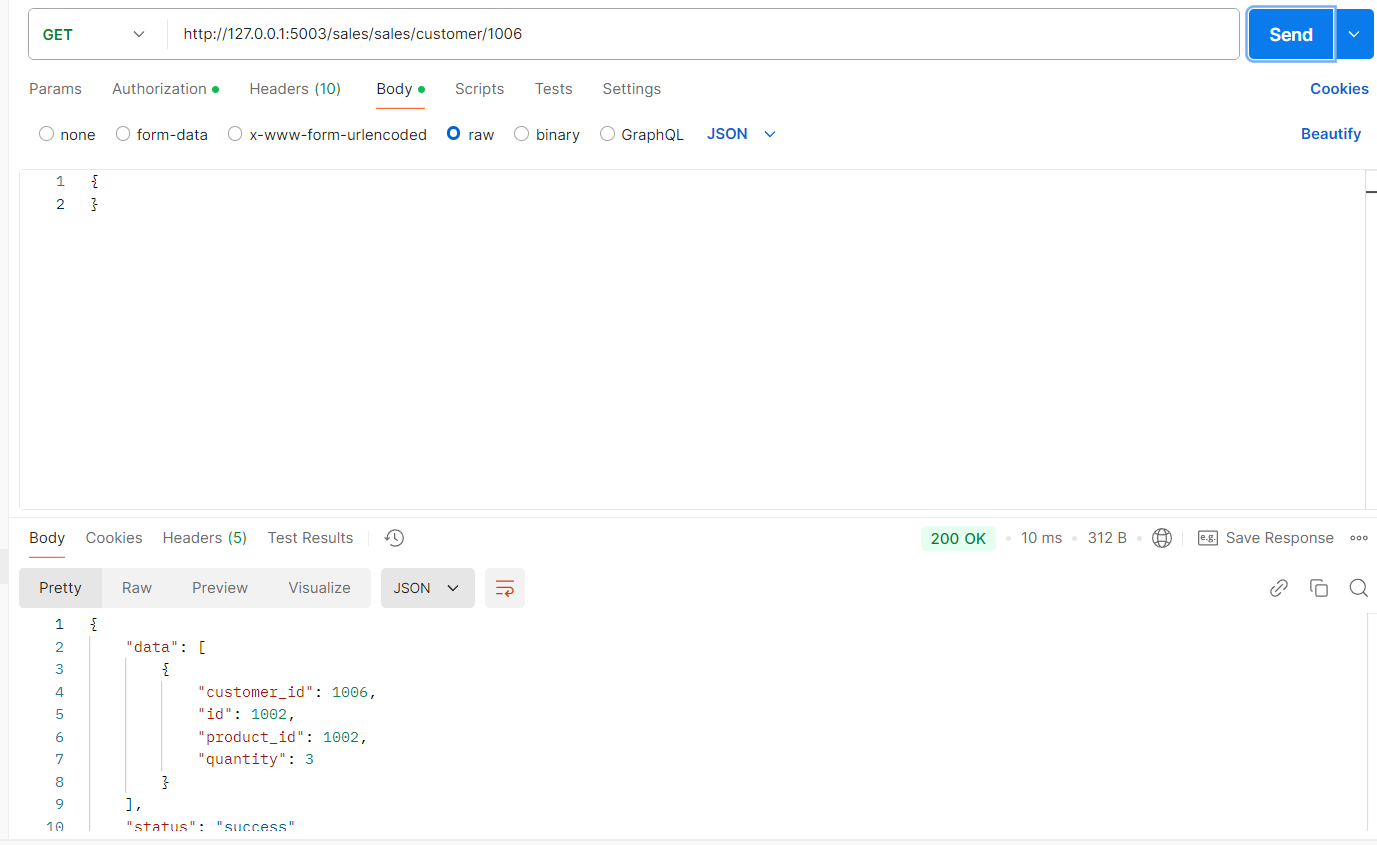
Description automatically generated

Get all sales:

A screenshot of a computer

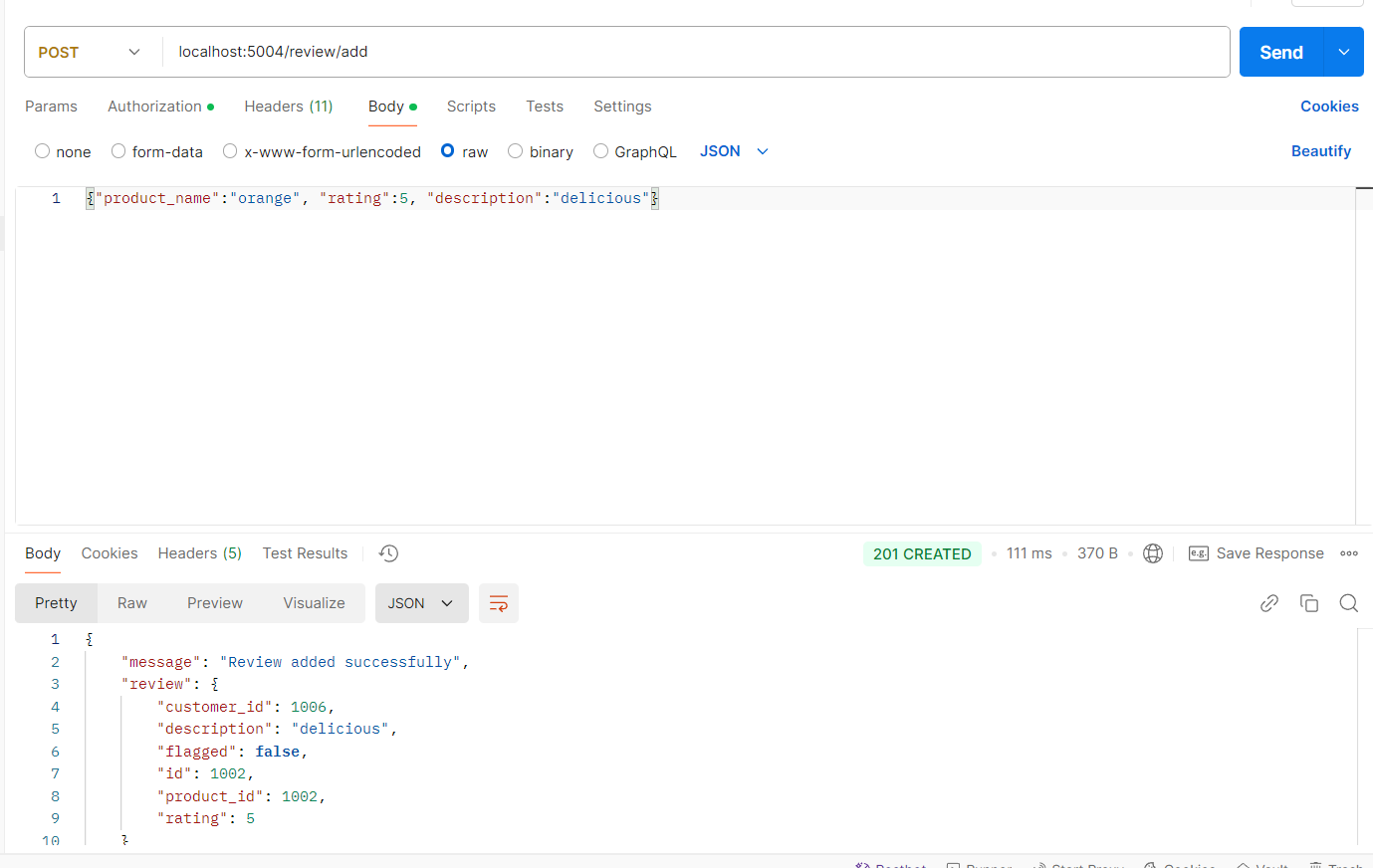
Description automatically generated

Get sales by customer:



**Postman API calls- Review\_service:**

Add review (needs authorization and the customer must have bought this product):

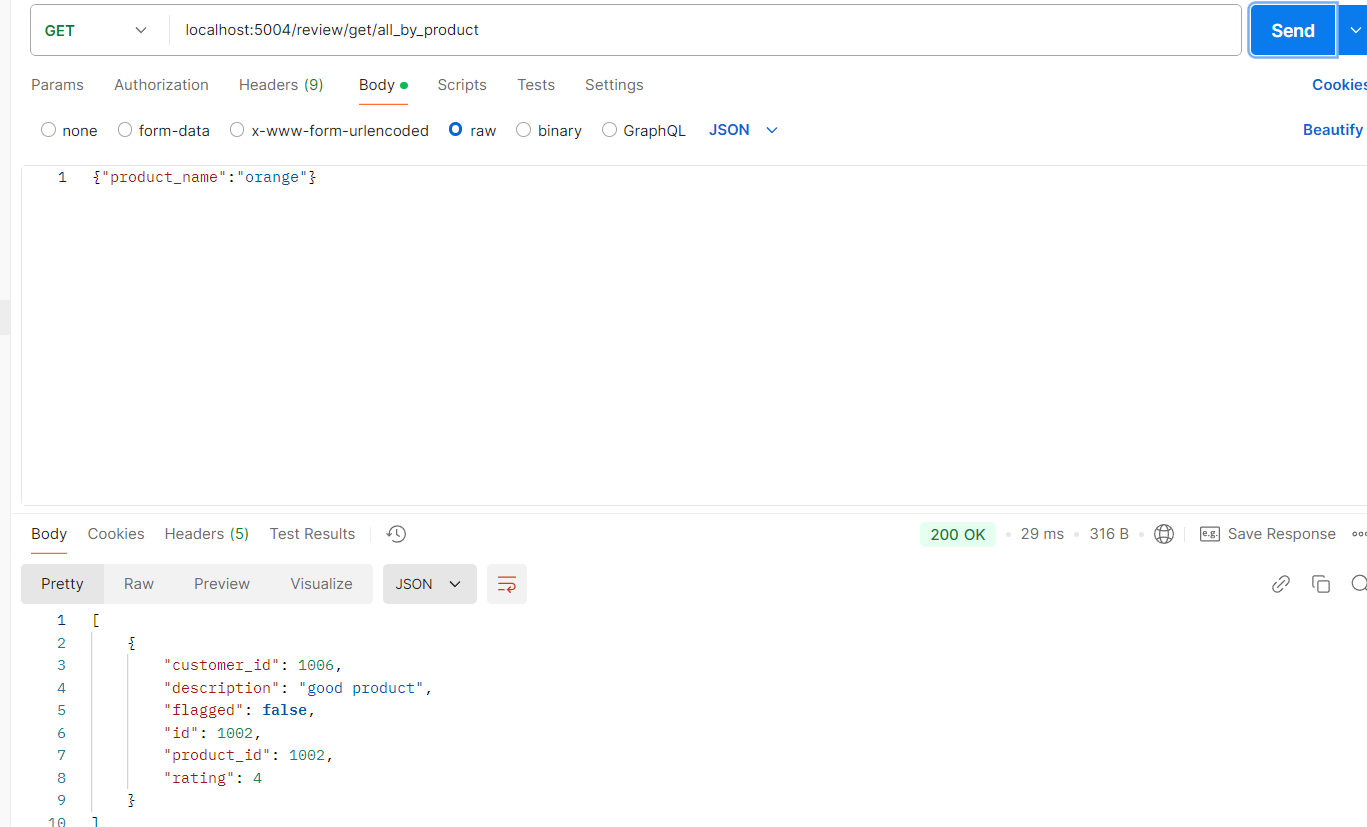


Update review (needs authorization):

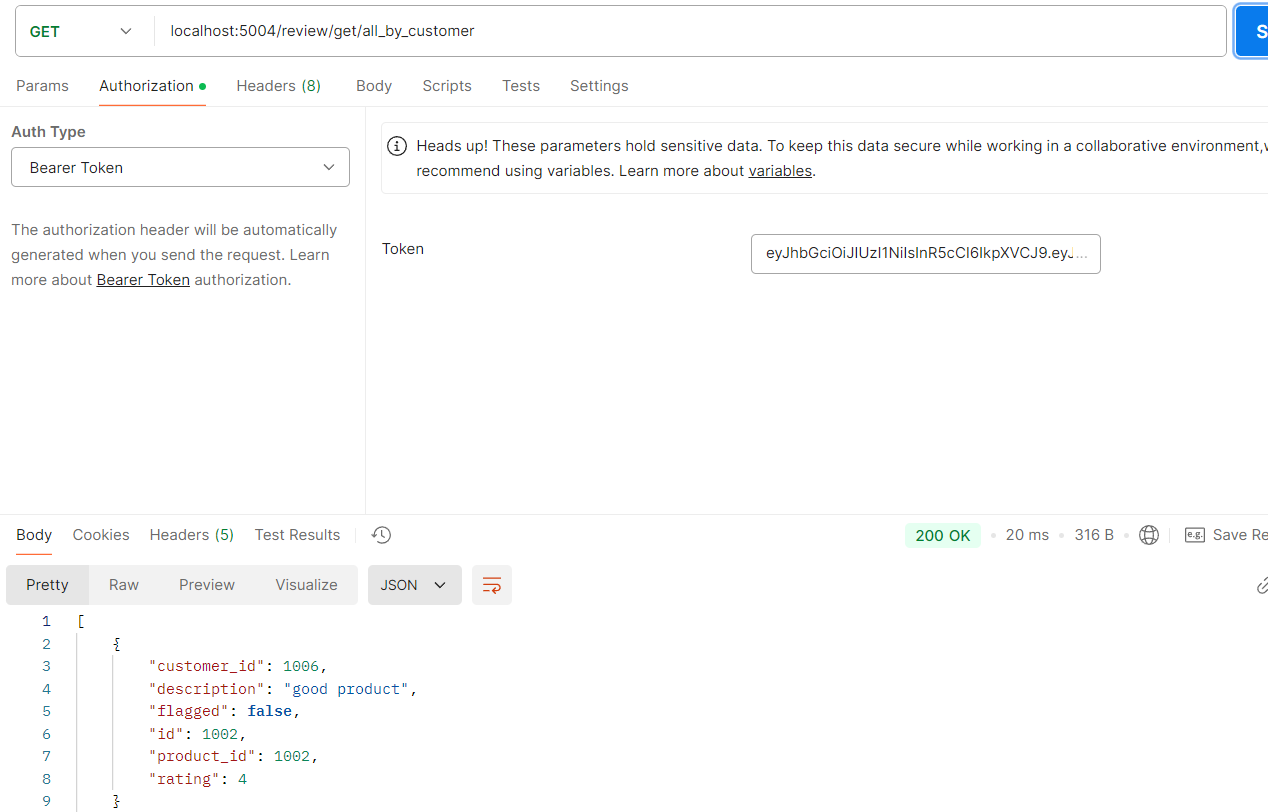
A screenshot of a computer

Description automatically generated

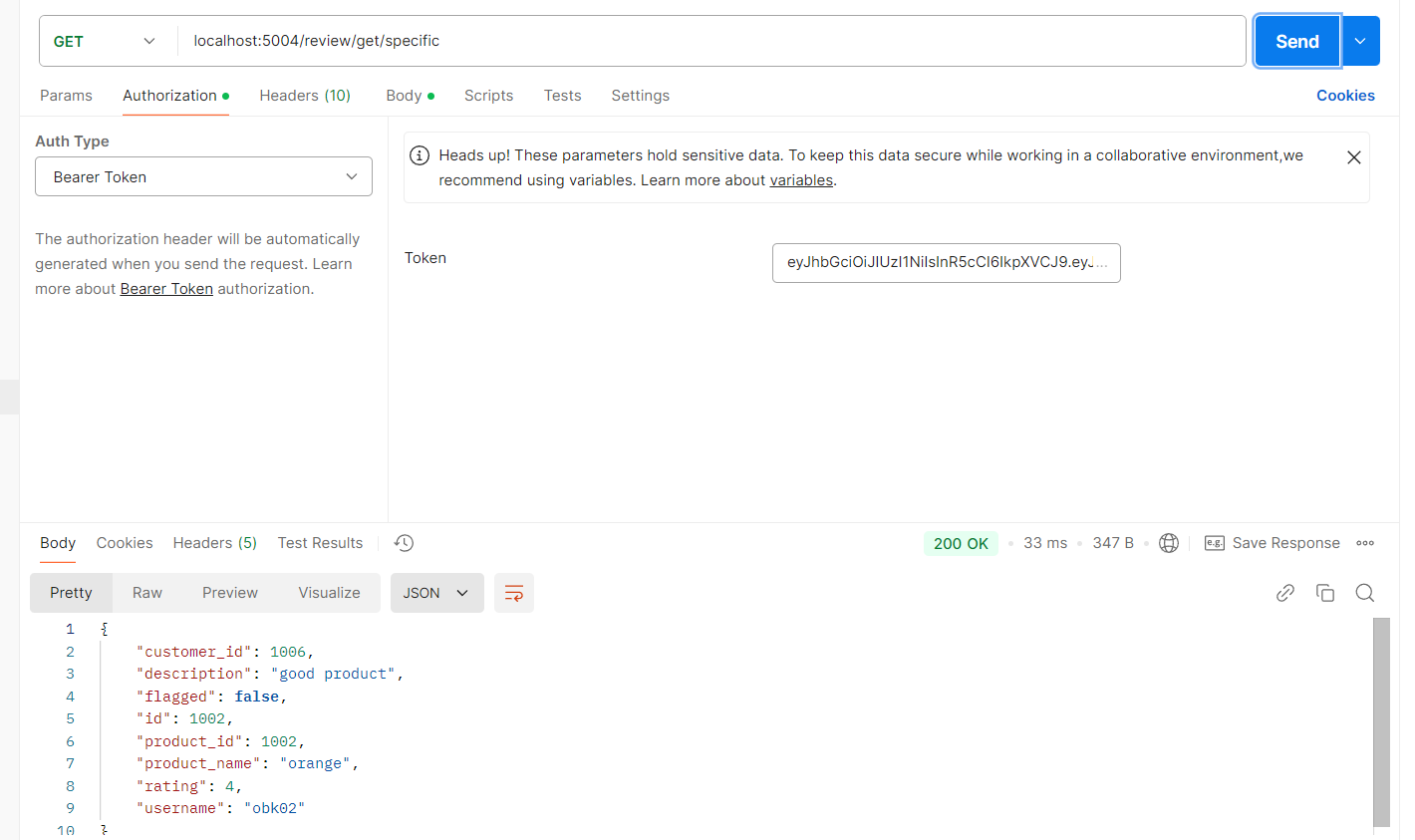
Get all reviews for a specific product:



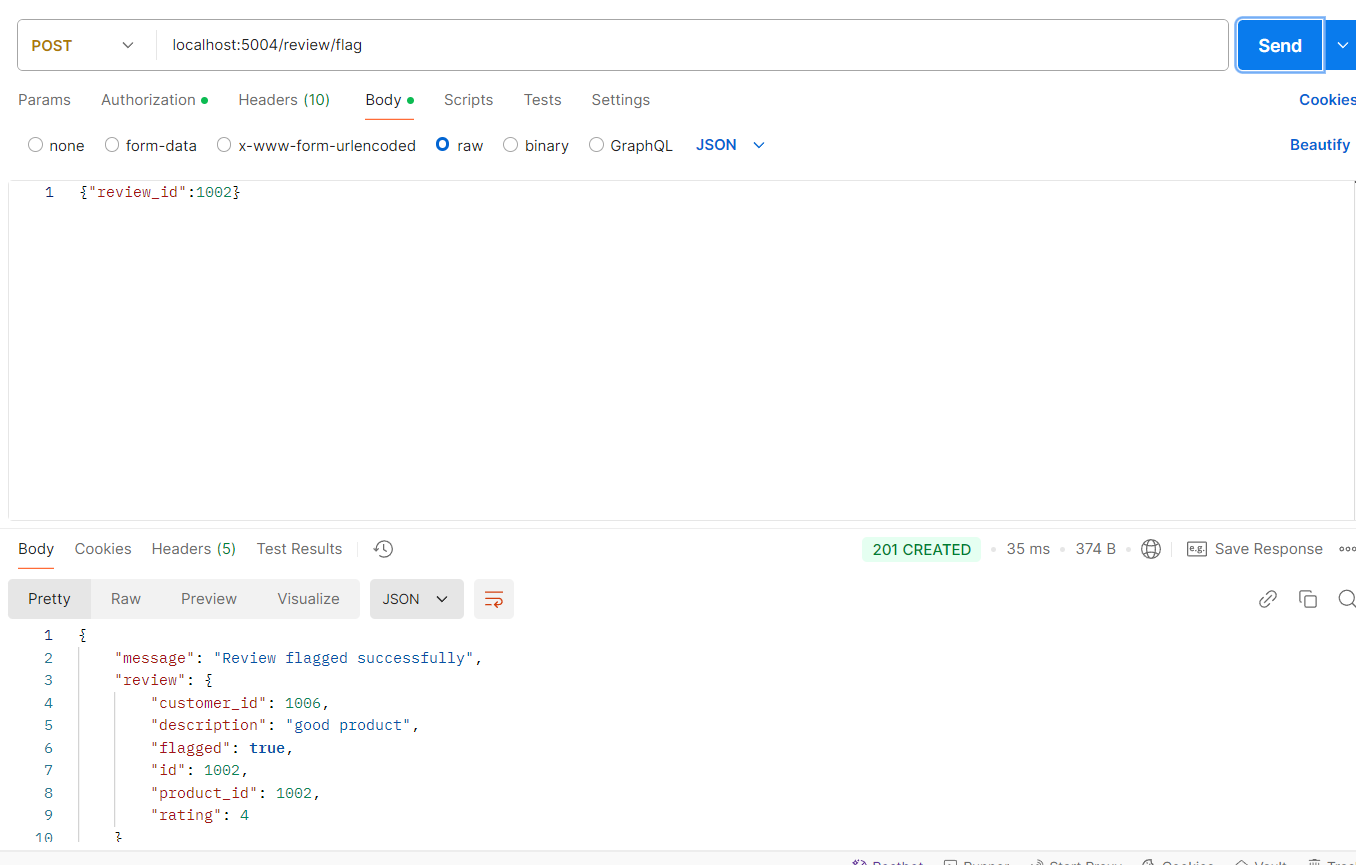
Get all reviews for a specific customer:



Get details of a specific review (admin only):



Flag review (admin only and flagged reviews don’t appear in get functions anymore):



Admin deleting review:

A screenshot of a computer

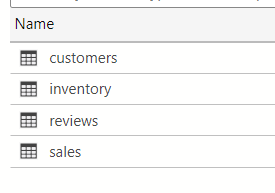
Description automatically generated

Customer deleting their own review:

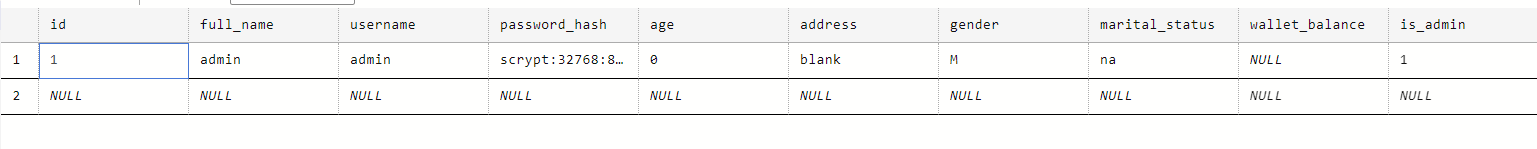
A screenshot of a computer

Description automatically generated

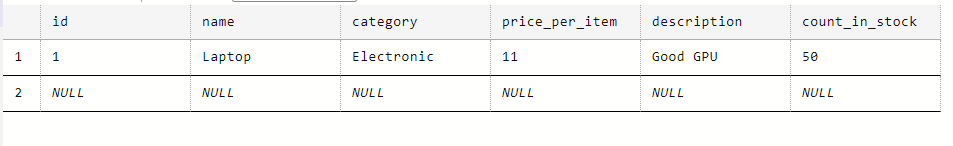
Database:



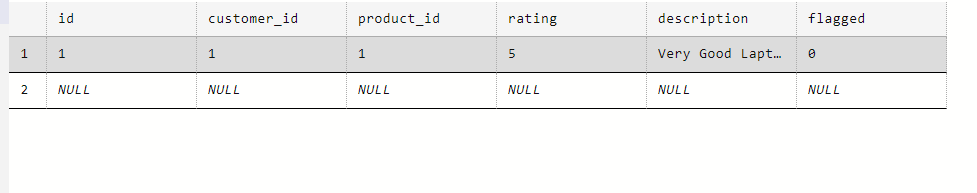
Customers Table



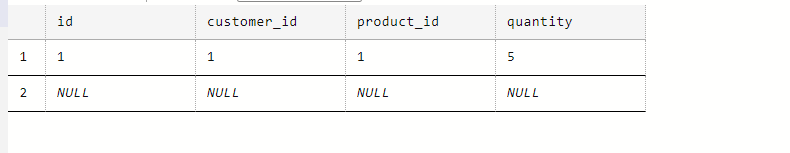
Inventory Table:



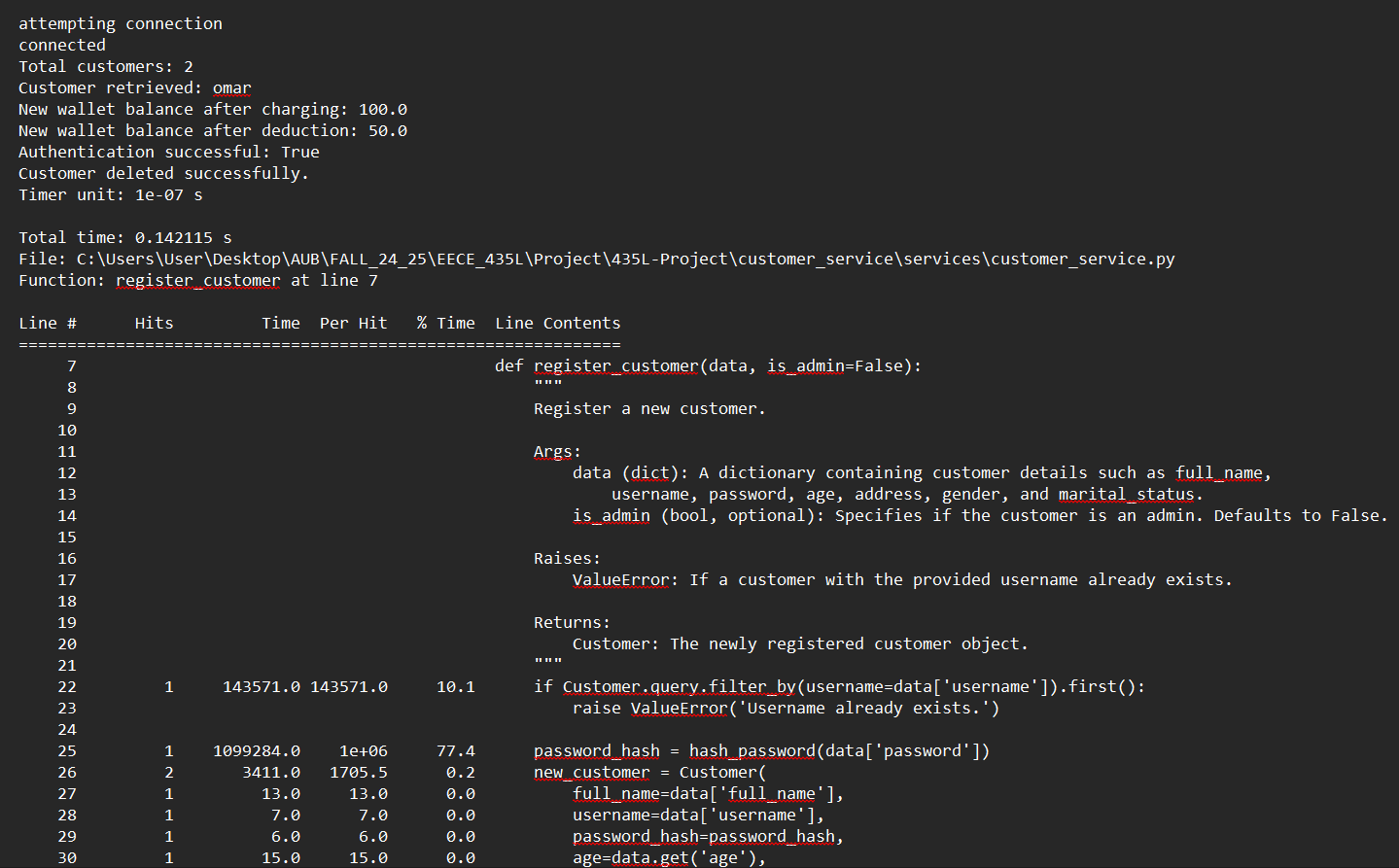
Review Table:

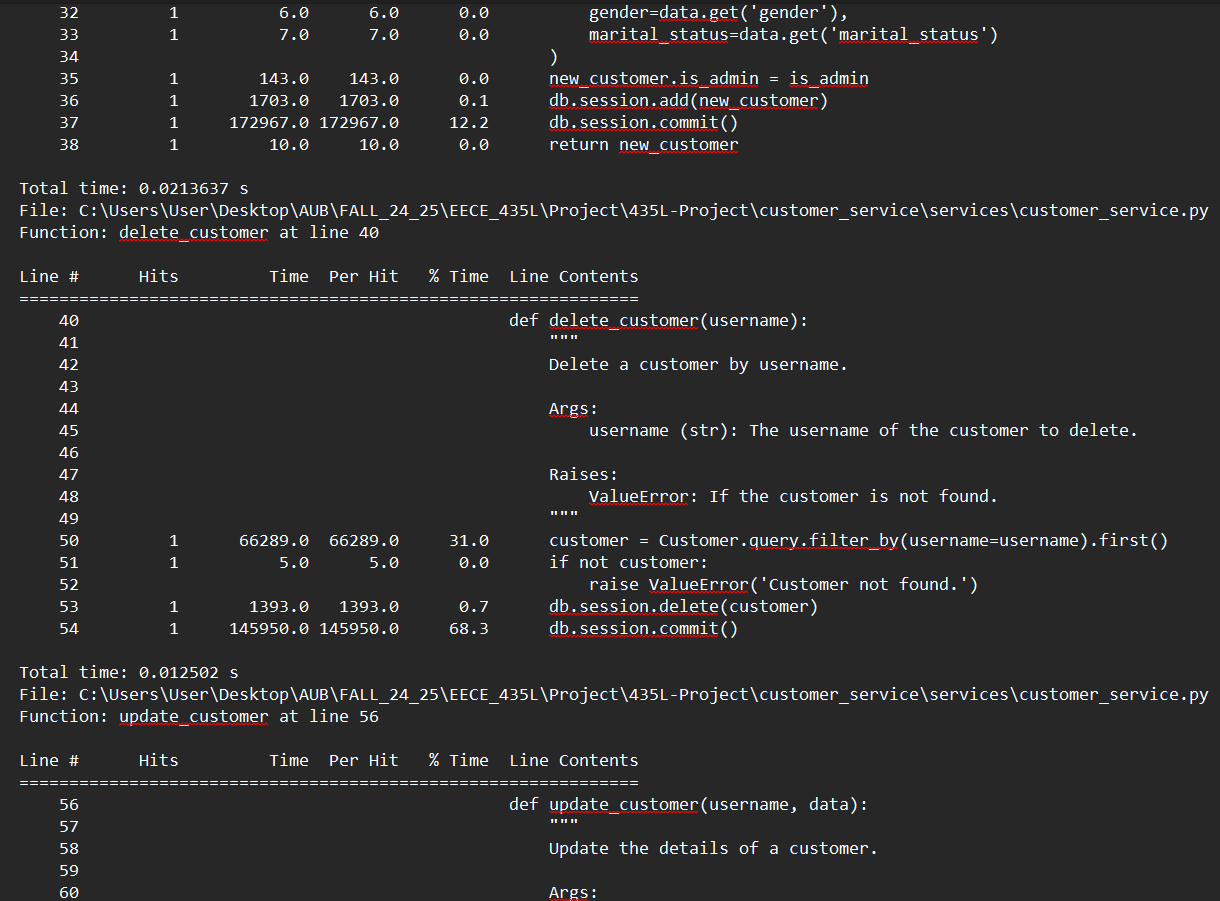


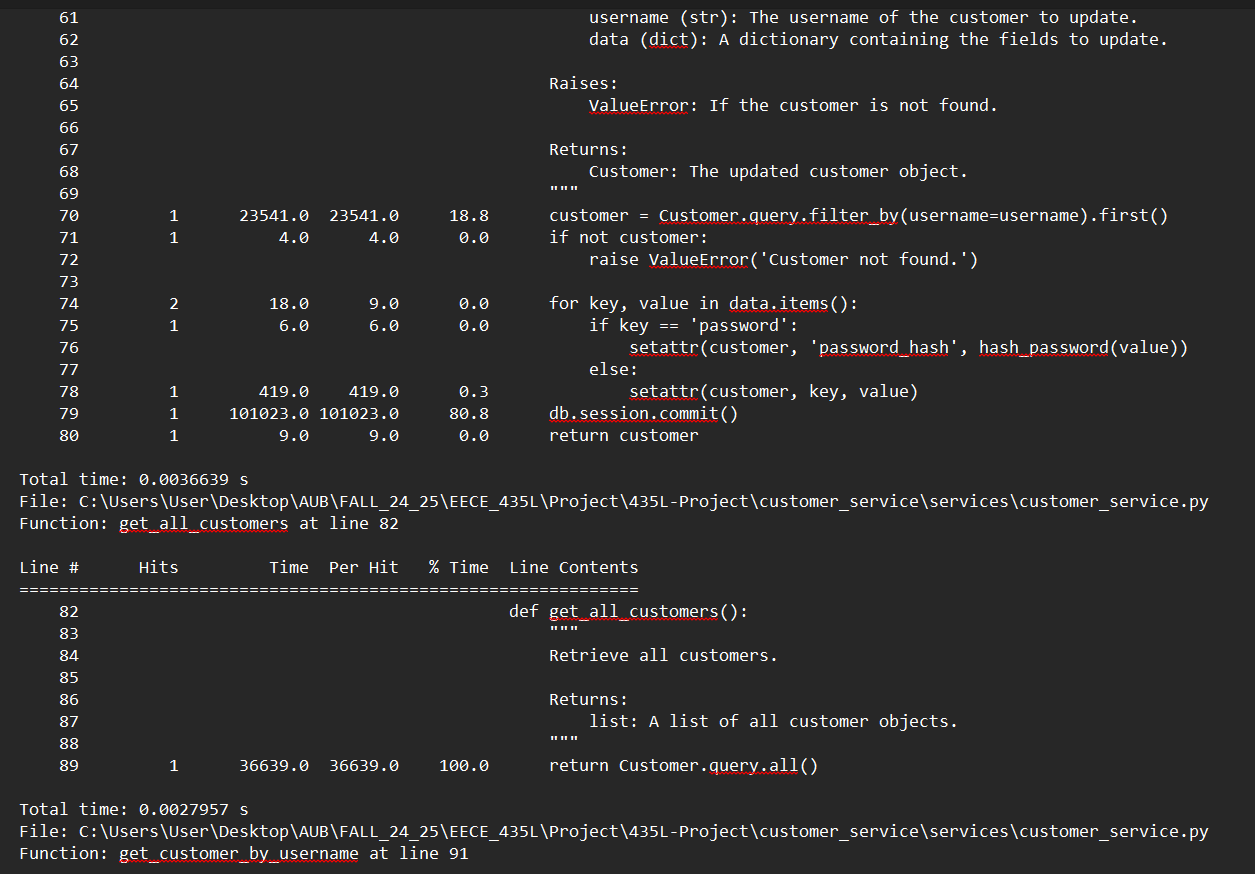
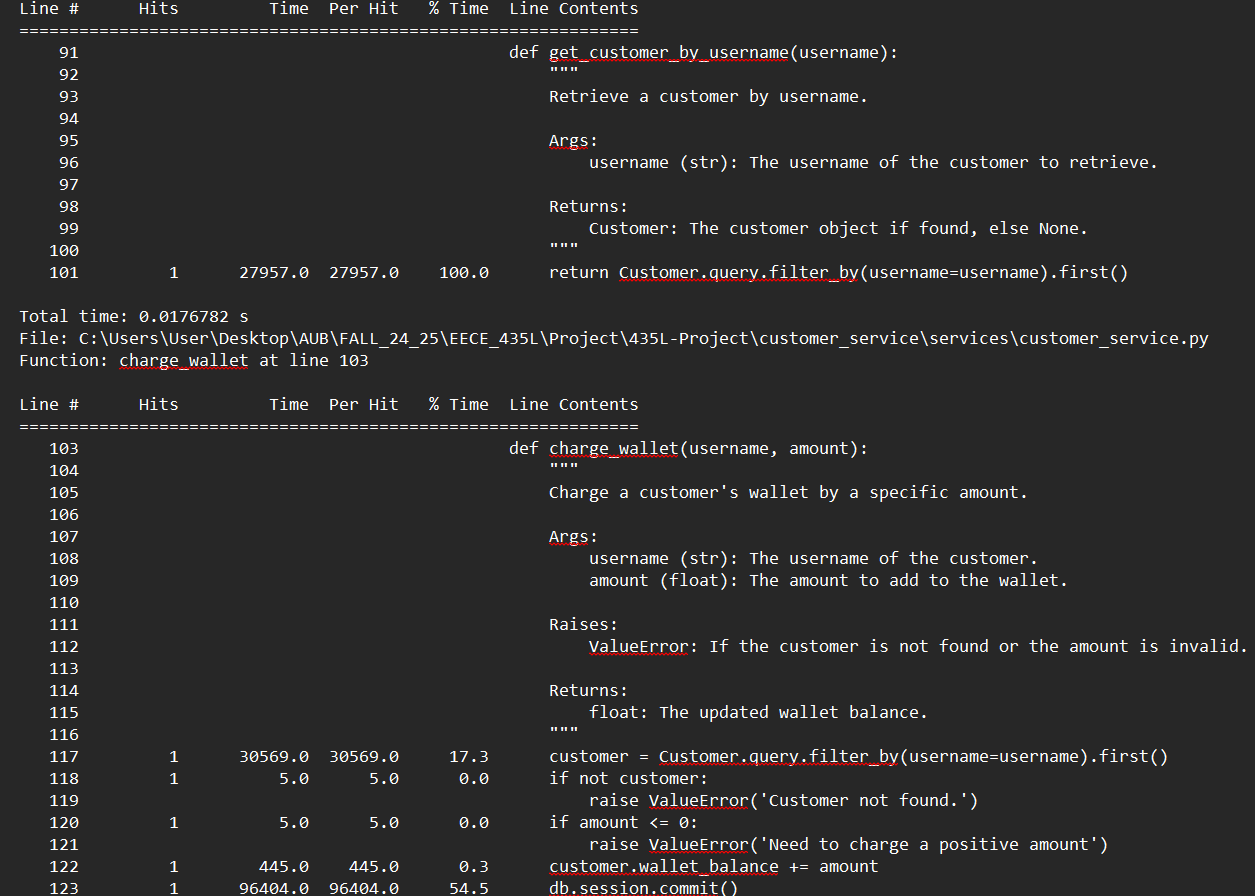
Sales Table:

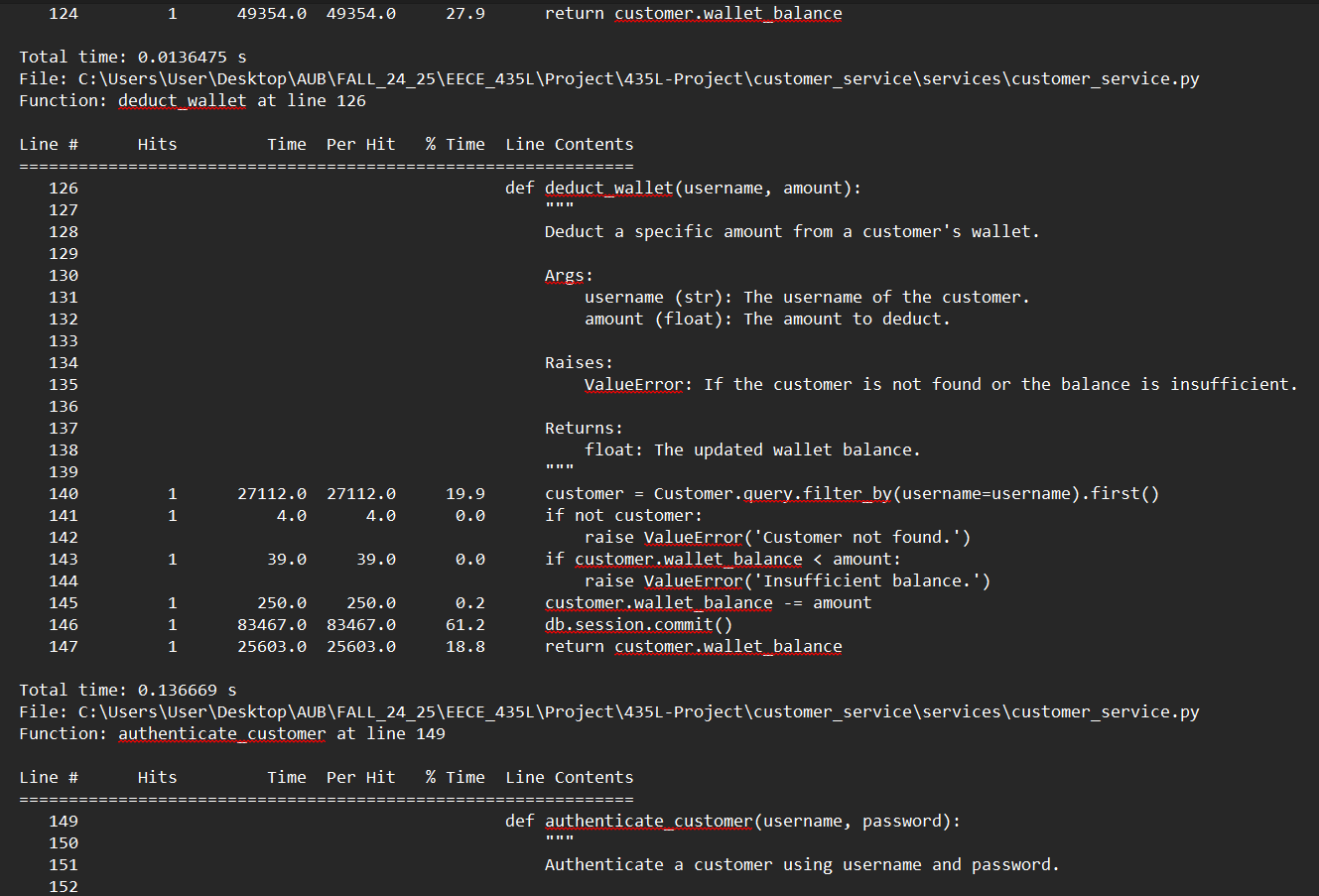


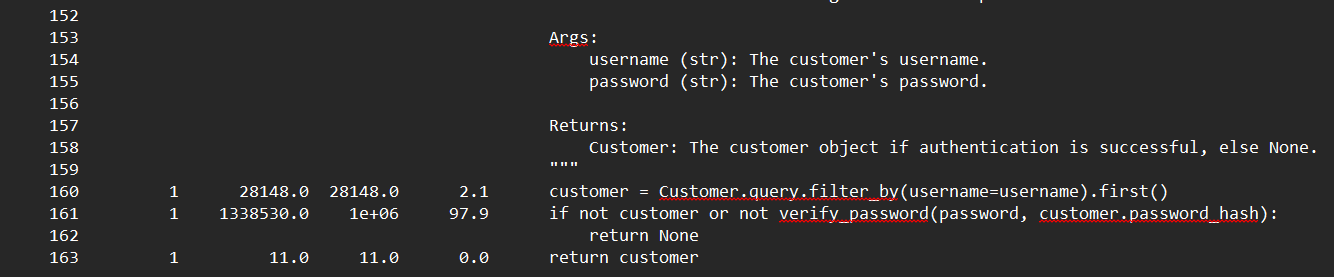
Customer Line Profiler:





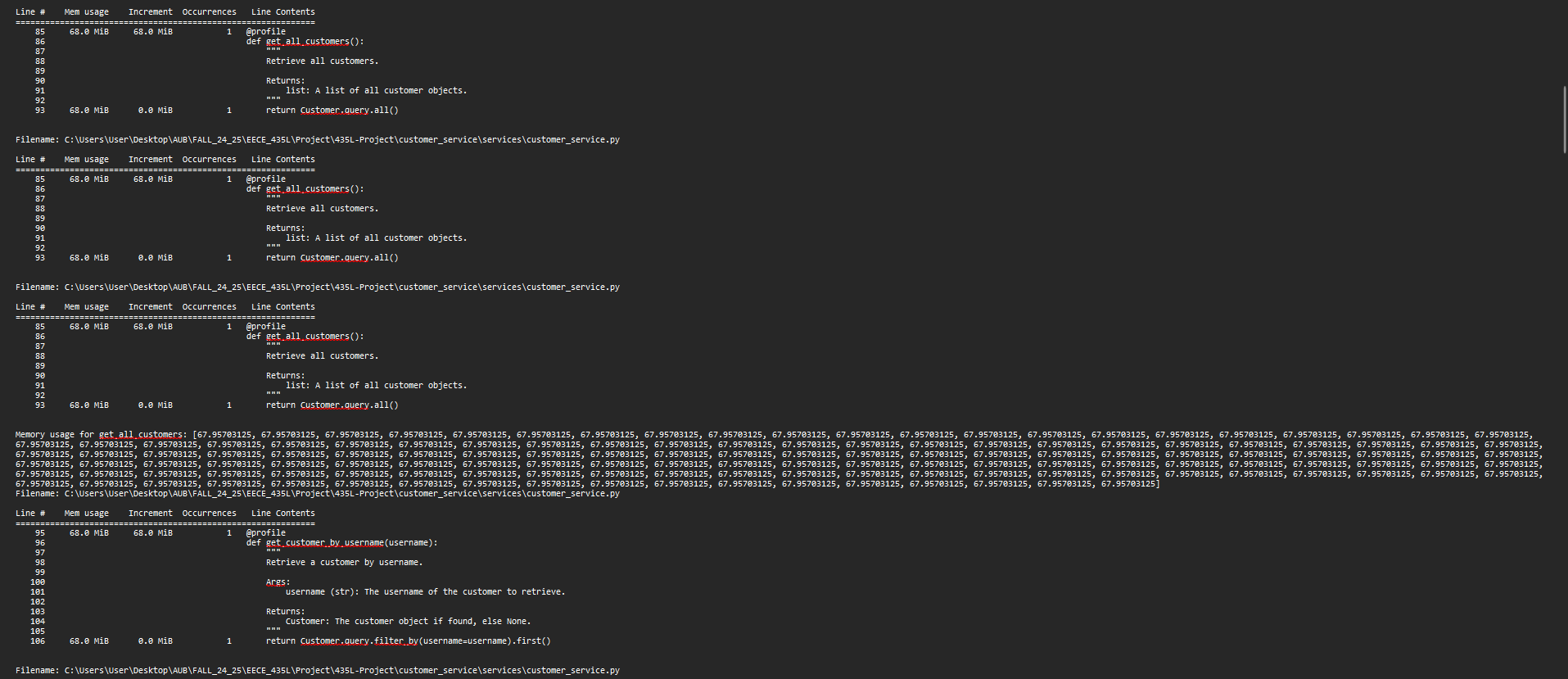
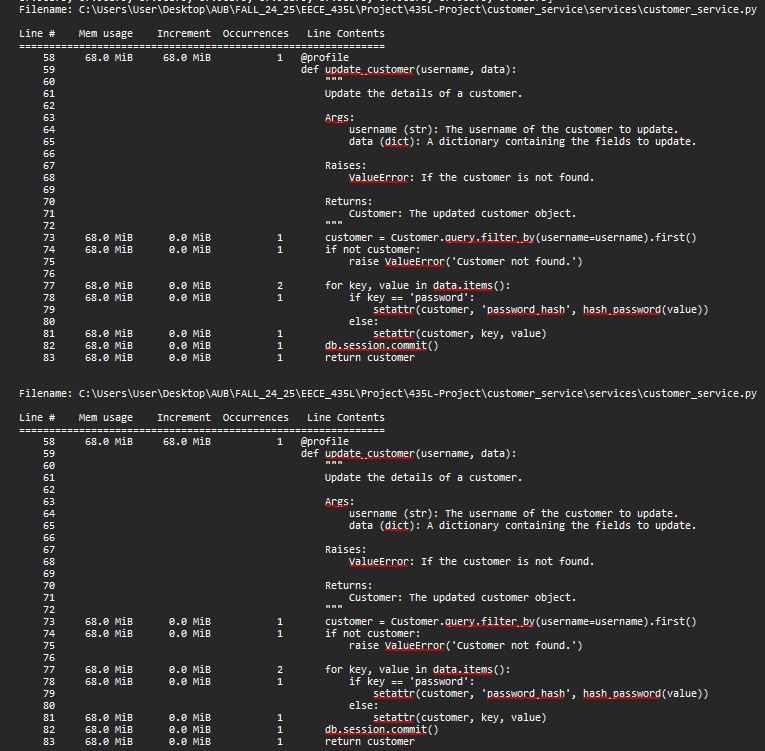
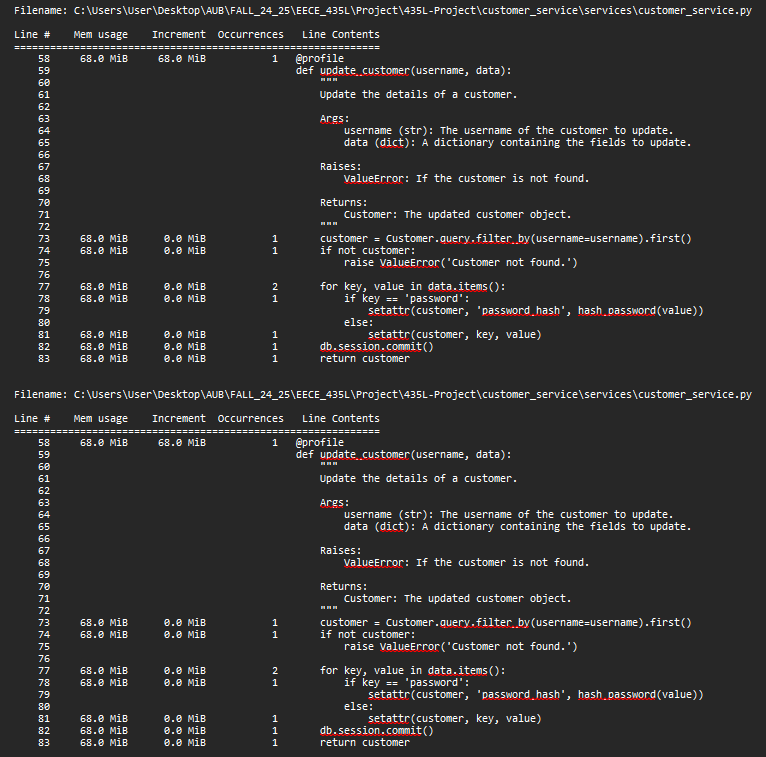
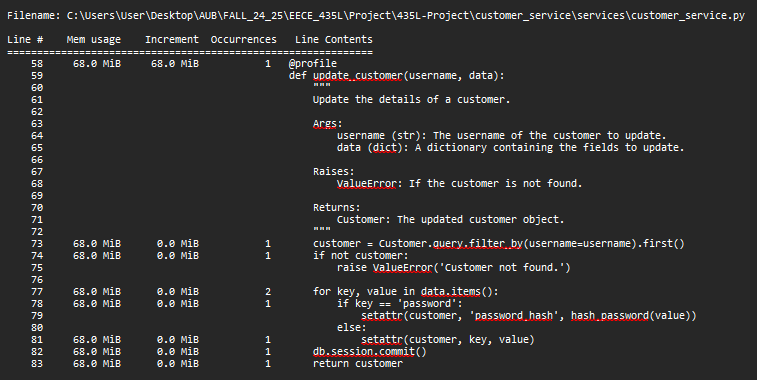
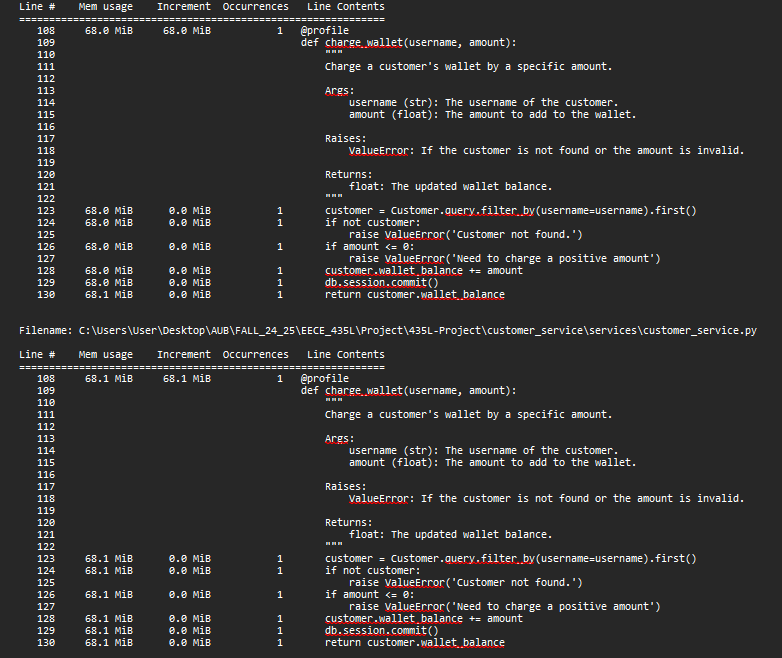
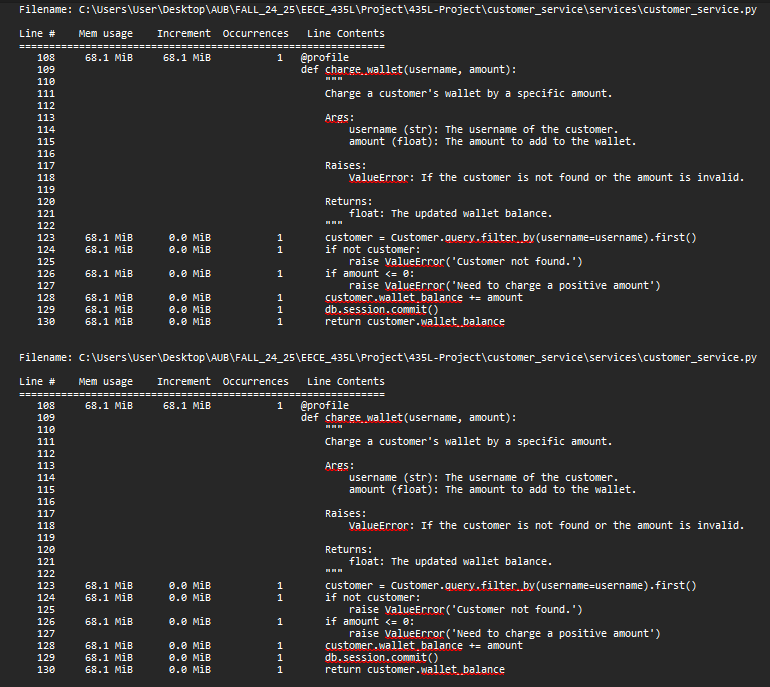
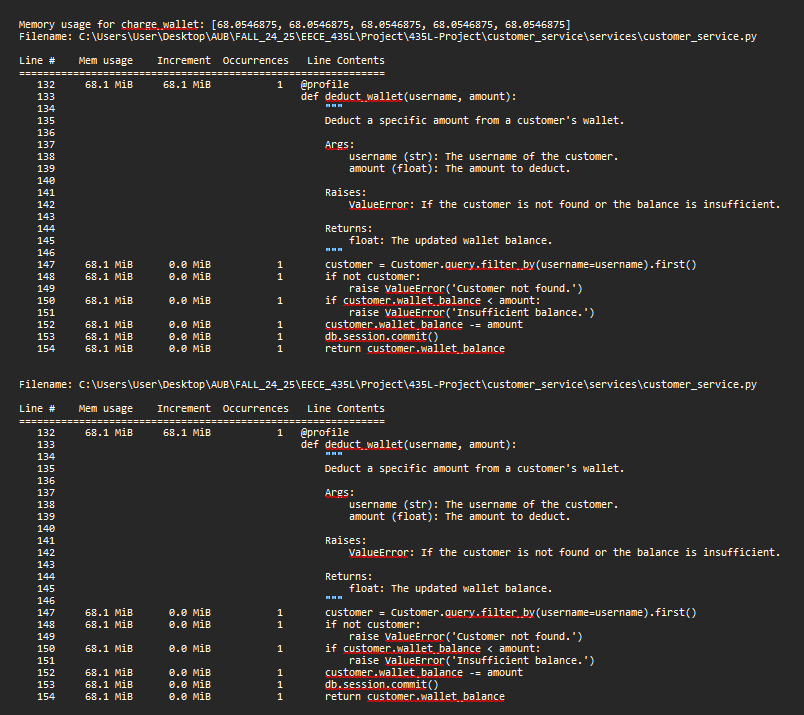
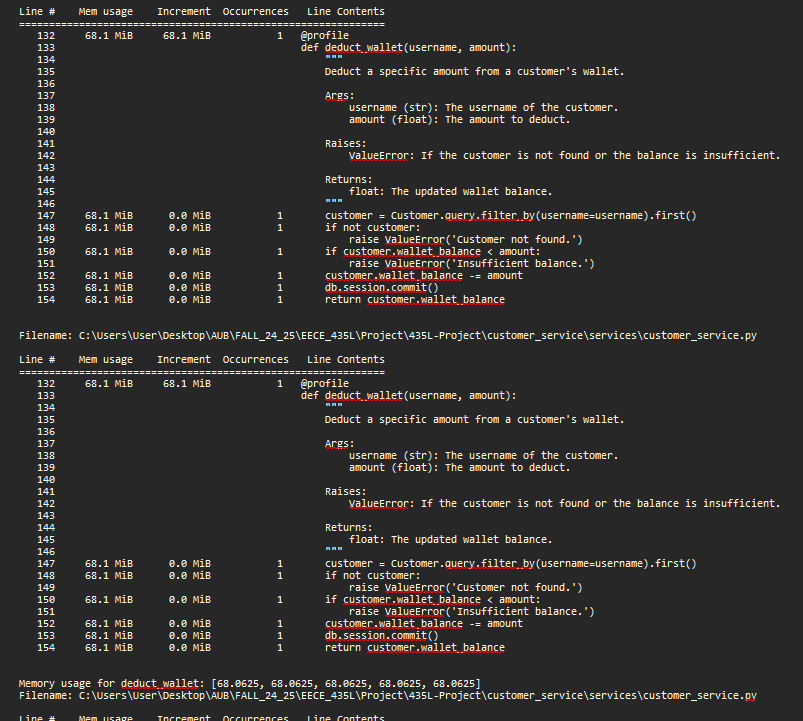
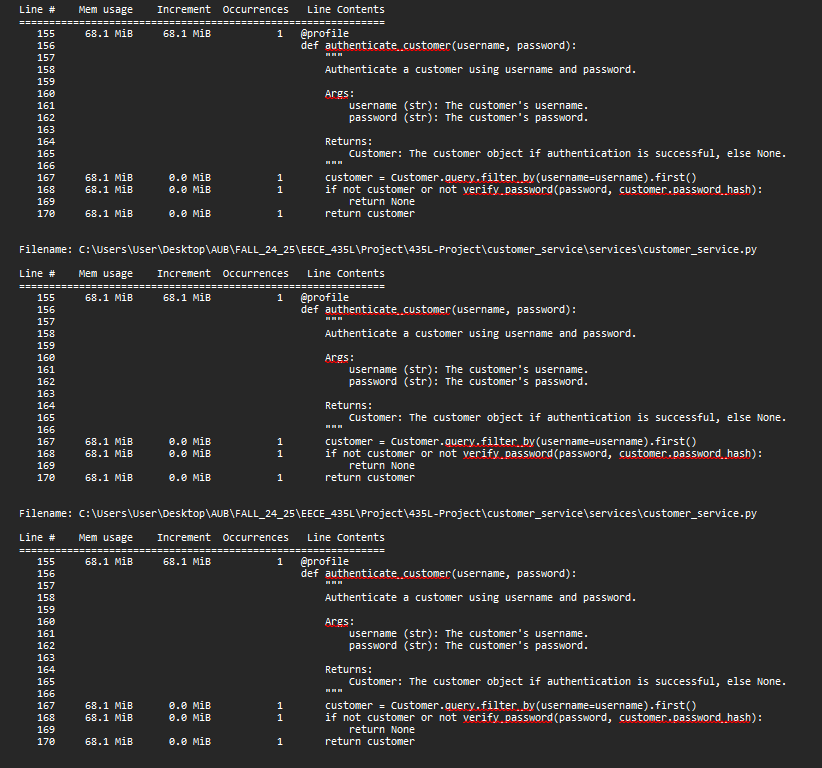
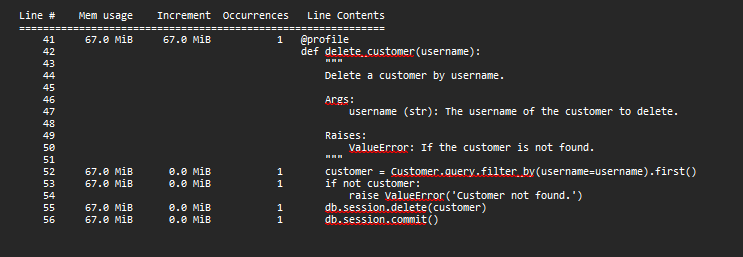
 



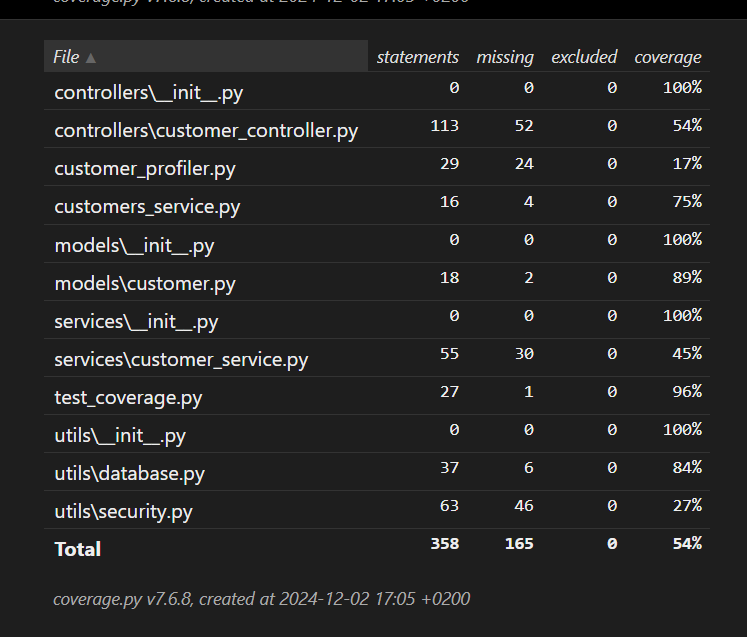


Memory Profile Customer Service



Customer Coverage:



Similar Profiling and coverage can be found in the folders of each service respectively. Line profile are found in text files called line\_profiler and memory\_profiler.

The coverage html files are found in htmlcov index file.

Running the Dockers compose command:

