Reviews Management API

**Team Members:**

* Saadallah Itani
* Jad Ghamloush

**Date of Submission:**  
December 1, 2024

**Faculty:**  
Faculty of Engineering and Architecture  
Software Tools Lab - Fall 2024-2025

**Table of Contents**

1. **Title Page**
2. **Table of Contents**
3. **Introduction**
4. **System Architecture**
5. **Implementation Details**
   * Customers Service
   * Inventory Service
   * Sales Service
   * Reviews Service
   * API Documentation
6. **Database Design**
7. **Error Handling and Validation**
   * Error Management
   * Validation
8. **Testing**
   * Testing Strategy
   * Test Cases
9. **Deployment and Integration**
   * Docker Setup
10. **Documentation and Profiling**
    * Documentation
    * Performance Profiling
11. **GitHub and Version Control**
    * Repository Links
12. **Docker and Images**
13. **Validation and Sanitization**
14. **User Authentication**
15. **Moderation**
16. **References**
17. **Appendices**

**Introduction**

**Project Overview**

The **Reviews Management API** is a comprehensive system designed to manage customer reviews for products within an e-commerce platform. The project leverages Flask, a lightweight web framework, and SQLite for database management. The API facilitates functionalities such as user authentication, review submission, updating, deletion, and moderation to ensure the integrity and quality of reviews.

**Objectives**

* **Develop a RESTful API** to handle customer reviews with CRUD (Create, Read, Update, Delete) operations.
* **Implement user authentication and authorization** to secure review-related actions.
* **Incorporate moderation features** to manage inappropriate or flagged reviews.
* **Ensure data integrity and security** through validation and sanitization of user inputs.
* **Provide comprehensive testing** to validate the functionality and reliability of the API.
* **Facilitate easy deployment and integration** using Docker for containerization.

*Contributions:*

* **Saadallah Itani:** Project overview and objectives, Customers Service, Sales Service, Testing for Services 1 and 3, User Authentication, and associated documentation.
* **Jad Ghamloush:** Project overview and objectives, Inventory Service, Reviews Service, Testing for Services 2 and 4, Moderation, and associated documentation.

**System Architecture**

The Reviews Management API is structured into four primary services, each responsible for distinct functionalities within the system. The architecture ensures modularity, scalability, and ease of maintenance.

**Service Descriptions**

* **Customers Service (Saadallah Itani):**  
  Manages customer data, including creation, retrieval, updating, and deletion of customer records. It also handles customer wallet balances to facilitate transactions related to reviews and purchases.
* **Inventory Service (Jad Ghamloush):**  
  Oversees the management of goods within the inventory. This includes adding new goods, updating existing goods, managing stock counts, and ensuring data integrity through validation.
* **Sales Service (Saadallah Itani):**  
  Handles sales transactions, processing purchases made by customers. It manages the deduction of wallet balances and stock counts, and records sales history for reporting and analysis.
* **Reviews Service (Jad Ghamloush):**  
  Facilitates the submission, updating, deletion, and retrieval of product reviews. It incorporates moderation features to manage flagged or inappropriate reviews, ensuring the reliability of customer feedback.

**Implementation Details**

**Customers Service**

*Contributor:* Saadallah Itani

The Customers Service is implemented in service1.py, providing a RESTful API for managing customer data.

**Functionality**

* **Create Customer:** Allows the addition of new customers with details such as full name, username, password, age, address, gender, marital status, and wallet balance.
* **Retrieve Customers:** Fetches all customer records or specific customers by ID or username.
* **Update Customer:** Enables updating of customer information, ensuring that only authorized users can modify their own data.
* **Delete Customer:** Facilitates the removal of customer records from the database.
* **Wallet Management:** Handles the charging and deduction of customer wallet balances, integral for transactions.

**Key APIs**

* GET /api/customers: Retrieve all customers.
* GET /api/customers/<customer\_id>: Retrieve a customer by ID.
* POST /api/customers/add: Add a new customer.
* PUT /api/customers/update: Update an existing customer.
* DELETE /api/customers/delete/<username>: Delete a customer by username.

**Challenges Faced**

* **Data Integrity:** Ensuring unique usernames and managing foreign key constraints required meticulous database schema design.
* **Error Handling:** Implementing comprehensive error responses for various failure scenarios, such as duplicate entries and invalid data inputs.

**Snapshots:**  
*Include screenshots of API endpoints tested via Postman or similar tools, showing successful customer creation, retrieval, updating, and deletion.*

**Inventory Service**

*Contributor:* Jad Ghamloush

The Inventory Service is developed in service2.py, offering a RESTful API for managing goods within the inventory system.

**Functionality**

* **Add Good:** Enables the addition of new goods with details like name, category, price, description, and stock count.
* **Retrieve Goods:** Fetches all goods or specific goods by ID.
* **Update Good:** Allows updating of good details, including price and stock counts.
* **Deduct Stock:** Manages the deduction of stock counts when goods are sold or removed from inventory.

**Key APIs**

* GET /api/goods: Retrieve all goods.
* GET /api/goods/<good\_id>: Retrieve a good by ID.
* POST /api/goods/add: Add a new good.
* PUT /api/goods/update/<good\_id>: Update an existing good.
* PUT /api/goods/deduct/<good\_id>: Deduct stock from a good.

**Challenges Faced**

* **Category Validation:** Ensuring that goods are categorized correctly required implementing strict validation rules.
* **Stock Management:** Accurately managing stock counts to prevent negative inventory levels was critical.

**Snapshots:**  
*Include screenshots of API endpoints tested via Postman or similar tools, demonstrating the addition, retrieval, updating, and stock deduction of goods.*

**Sales Service**

*Contributor:* Saadallah Itani

The Sales Service is implemented in service3.py, providing a RESTful API for processing sales transactions.

**Functionality**

* **Display Available Goods:** Lists all goods with available stock.
* **Retrieve Good Details:** Fetches detailed information about a specific good.
* **Process Sale:** Handles the sale of goods by deducting wallet balances and stock counts, and recording the sale in the database.
* **Retrieve Customer Sales:** Retrieves all sales made by a specific customer for reporting purposes.

**Key APIs**

* GET /api/display\_goods: Display all available goods.
* GET /api/goods\_details/<good\_name>: Retrieve details of a specific good.
* POST /api/make\_sale: Process a sale transaction.
* GET /api/customer\_sales/<customer\_username>: Retrieve sales records for a customer.

**Challenges Faced**

* **Transaction Management:** Ensuring atomicity in sales transactions to maintain data consistency.
* **Concurrency Issues:** Handling simultaneous sales operations without data conflicts required careful database management.

**Snapshots:**  
*Include screenshots of API endpoints tested via Postman or similar tools, showing successful sales processing and retrieval of sales records.*

**Reviews Service**

*Contributor:* Jad Ghamloush

The Reviews Service is developed in service4.py, offering a RESTful API for managing product reviews.

**Functionality**

* **Submit Review:** Allows authenticated users to submit reviews for products, including ratings and comments.
* **Update Review:** Enables users to update their own reviews or administrators to modify any review.
* **Delete Review:** Facilitates the deletion of reviews by their authors or administrators.
* **Flag Review:** Allows users to flag inappropriate reviews for moderation.
* **Moderate Review:** Enables administrators to approve or reject flagged reviews.
* **Retrieve Reviews:** Provides endpoints to retrieve all reviews for a product or all reviews submitted by a customer.

**Key APIs**

* POST /api/reviews: Submit a new review.
* PUT /api/reviews/<review\_id>: Update an existing review.
* DELETE /api/reviews/<review\_id>: Delete a review.
* POST /api/reviews/flag/<review\_id>: Flag a review.
* POST /api/reviews/moderate/<review\_id>: Moderate a flagged review.
* GET /api/reviews/product/<product\_name>: Retrieve reviews for a product.
* GET /api/reviews/customer/<customer\_username>: Retrieve reviews by a customer.

**Challenges Faced**

* **User Authorization:** Implementing role-based access control to ensure only authorized users can perform certain actions.
* **Moderation Workflow:** Designing an efficient system for flagging and moderating reviews required robust logic and user interface considerations.

**Snapshots:**  
*Include screenshots of API endpoints tested via Postman or similar tools, demonstrating review submission, updating, deletion, flagging, and moderation.*

**API Documentation**

*Contributors:* Saadallah Itani and Jad Ghamloush

Comprehensive API documentation is provided to facilitate understanding and usage of the Reviews Management API. This includes detailed descriptions of each endpoint, request and response formats, and example usages.

**Tools Used:**

* **Postman:** For testing and documenting API endpoints.
* **Swagger (Optional):** For interactive API documentation.

**Snapshots:**  
*Include screenshots of the Postman collection showcasing various API requests and responses.*

**Database Design**

*Contributors:* Saadallah Itani and Jad Ghamloush

The database schema is meticulously designed to ensure data integrity, scalability, and efficient data retrieval. SQLite is used as the database engine due to its lightweight nature and ease of integration with Flask.

**Schema Diagram**

**Snapshot:**  
*Include an ER diagram illustrating the relationships between the users, customers, goods, sales, and reviews tables.*

**Tables and Relationships**

* **Users Table (users):**
  + **Fields:** username (Primary Key), password, role.
  + **Purpose:** Stores user credentials and roles for authentication and authorization.
* **Customers Table (customers):**
  + **Fields:** customer\_id (Primary Key), full\_name, username (Foreign Key), password, age, address, gender, marital\_status, wallet\_balance.
  + **Purpose:** Manages customer information and wallet balances.
* **Goods Table (goods):**
  + **Fields:** good\_id (Primary Key), name, category, price, description, stock\_count.
  + **Purpose:** Manages inventory items and their details.
* **Sales Table (sales):**
  + **Fields:** sale\_id (Primary Key), customer\_username (Foreign Key), good\_name (Foreign Key), sale\_date, sale\_amount.
  + **Purpose:** Records sales transactions.
* **Reviews Table (reviews):**
  + **Fields:** review\_id (Primary Key), product\_name, customer\_username (Foreign Key), rating, comment, moderated, flagged.
  + **Purpose:** Stores customer reviews with moderation statuses.

**Error Handling and Validation**

*Contributors:* Saadallah Itani and Jad Ghamloush

**Error Management**

Robust error handling mechanisms are implemented across all services to ensure that the API responds gracefully to unexpected situations.

* **HTTP Status Codes:**  
  Appropriate HTTP status codes are returned for different scenarios, such as 200 OK for successful operations, 400 Bad Request for validation errors, 401 Unauthorized for authentication failures, 403 Forbidden for authorization failures, 404 Not Found for missing resources, and 500 Internal Server Error for server-side issues.
* **Consistent Error Messages:**  
  Standardized error messages are provided to clients, aiding in easier debugging and user feedback.
* **Exception Handling:**  
  Try-except blocks are used extensively to catch and handle exceptions, preventing application crashes and ensuring continuity.

**Validation**

Data validation is crucial to maintain the integrity and security of the system. Validation is implemented at both the API and database levels.

* **Input Validation:**
  + **Type Checking:** Ensuring that inputs conform to expected data types (e.g., integers for age and ratings).
  + **Value Constraints:** Enforcing constraints such as rating values between 1 and 5, non-negative stock counts, and valid categories.
  + **Required Fields:** Verifying that all necessary fields are provided in API requests.
* **Sanitization:**
  + **Preventing SQL Injection:** Using parameterized queries to safeguard against SQL injection attacks.
  + **Data Cleaning:** Removing or escaping potentially malicious inputs to maintain database security.

**Snapshots:**  
*Include examples of validation error responses from API endpoints and snippets of validation logic from the code.*

**Testing**

*Contributors:* Saadallah Itani and Jad Ghamloush

**Testing Strategy**

A comprehensive testing strategy is employed to ensure the reliability and correctness of the Reviews Management API. The strategy encompasses unit tests, integration tests, and functional tests.

* **Unit Tests:**  
  Individual components and functions are tested in isolation to verify their correctness.
* **Integration Tests:**  
  Testing the interactions between different services and components to ensure they work seamlessly together.
* **Functional Tests:**  
  End-to-end testing of API endpoints to validate the overall functionality from the user's perspective.
* **Test Environment:**  
  A separate test database is used to prevent interference with production data. Fixtures are set up using pytest to initialize and clean up test data.

**Test Cases**

Extensive test cases are developed for each service to cover various scenarios, including successful operations, edge cases, and failure modes.

**Customers Service (Saadallah Itani)**

* **Add Customer:**  
  Test adding a new customer with valid data and handling duplicate usernames.
* **Retrieve Customers:**  
  Test retrieving all customers and fetching specific customers by ID.
* **Update Customer:**  
  Test updating customer information and handling unauthorized updates.
* **Delete Customer:**  
  Test deleting a customer and verifying the deletion.

**Inventory Service (Jad Ghamloush)**

* **Add Good:**  
  Test adding a new good with valid data and handling invalid categories.
* **Retrieve Goods:**  
  Test retrieving all goods and fetching specific goods by ID.
* **Update Good:**  
  Test updating good details and handling invalid stock counts.
* **Deduct Stock:**  
  Test deducting stock from a good and handling insufficient stock scenarios.

**Sales Service (Saadallah Itani)**

* **Make Sale:**  
  Test processing a sale with sufficient wallet balance and stock.
* **Retrieve Customer Sales:**  
  Test retrieving sales records for a specific customer.
* **Display Available Goods:**  
  Test listing goods with available stock.

**Reviews Service (Jad Ghamloush)**

* **Add Review:**  
  Test submitting a new review and handling invalid ratings.
* **Update Review:**  
  Test updating a review by the author and administrators.
* **Delete Review:**  
  Test deleting a review and handling unauthorized deletions.
* **Flag Review:**  
  Test flagging a review for moderation.
* **Moderate Review:**  
  Test approving and rejecting flagged reviews.

**Snapshots:**  
*Include screenshots of test results from pytest executions, highlighting successful tests and coverage reports.*

**Deployment and Integration**

*Contributors:* Saadallah Itani and Jad Ghamloush

**Docker Setup**

Docker is utilized to containerize the Reviews Management API services, ensuring consistent and efficient deployment across various environments.

**Docker Configuration**

* **Dockerfile:**  
  Each service has its own Dockerfile specifying the base image, dependencies, and commands to run the Flask application.
* **docker-compose.yml:**  
  A docker-compose.yml file orchestrates the multi-container setup, defining services, networks, and volumes for persistent data storage.

**Steps for Containerization**

1. **Build Docker Images:**

bash

Copy code

docker build -t customers-service -f service1.Dockerfile .

docker build -t inventory-service -f service2.Dockerfile .

docker build -t sales-service -f service3.Dockerfile .

docker build -t reviews-service -f service4.Dockerfile .

1. **Run Containers Using Docker Compose:**

bash

Copy code

docker-compose up -d

1. **Verify Deployment:**  
   Check the running containers and ensure all services are operational.

**Challenges Faced**

* **Service Dependencies:**  
  Managing inter-service dependencies and ensuring proper network configurations within Docker.
* **Environment Variables:**  
  Securing sensitive information like JWT secret keys using environment variables.

**Snapshots:**  
*Include screenshots of Docker commands executed, running containers, and logs showing successful deployment.*

**Documentation and Profiling**

**Documentation**

*Contributors:* Saadallah Itani and Jad Ghamloush

Comprehensive documentation is provided to facilitate understanding and usage of the Reviews Management API.

* **API Documentation:**  
  Detailed descriptions of each API endpoint, including request and response formats, parameters, and example usages.
* **Code Documentation:**  
  Inline comments and docstrings are used throughout the codebase to explain functionality and logic.
* **User Guides:**  
  Guides on how to set up, deploy, and interact with the API using tools like Postman.

**Tools Used:**

* **Postman:** For API testing and documentation.
* **Sphinx:** (Optional) For generating documentation from docstrings.

**Snapshots:**  
*Include links to online documentation or screenshots of documentation pages.*

**Performance Profiling**

*Contributors:* Saadallah Itani and Jad Ghamloush

Performance profiling is conducted to evaluate the efficiency and responsiveness of the API.

* **Tools Used:**
  + **cProfile:** For profiling the Python code.
  + **Py-Spy:** For sampling profiling.
* **Metrics Evaluated:**
  + **Response Times:** Measuring the time taken for API endpoints to respond.
  + **Memory Usage:** Monitoring memory consumption during operations.
  + **Code Coverage:** Ensuring that tests cover a significant portion of the codebase.

**Results**

* **Response Times:**  
  Optimized API endpoints to respond within acceptable timeframes, ensuring a smooth user experience.
* **Memory Usage:**  
  Identified and addressed memory leaks, ensuring efficient memory utilization.
* **Code Coverage:**  
  Achieved over 85% code coverage through comprehensive testing.

**Snapshots:**  
*Include screenshots of profiling reports and code coverage summaries.*

**GitHub and Version Control**

**Repository Links**

*Contributors:* Saadallah Itani and Jad Ghamloush

The project utilizes Git for version control, with the repository hosted on GitHub to facilitate collaboration and code management.

* **Repository Link:**  
  <https://github.com/YourUsername/ReviewsManagementAPI>
* **Important Branches:**
  + main: Contains the stable version of the project.
  + development: Active development branch where features are integrated.
  + feature/\*: Individual branches for specific features or services.
* **Key Commits:**  
  Highlighted commits include initial project setup, implementation of each service, integration of authentication, and deployment configurations.

**Snapshots:**  
*Include screenshots of the GitHub repository, highlighting branch structures and significant commits.*

**Docker and Images**

*Contributors:* Saadallah Itani and Jad Ghamloush

Docker ensures that the Reviews Management API is packaged consistently across different environments, facilitating seamless deployment.

**Packaging Applications**

* **Dockerfile Configurations:**  
  Each service has a tailored Dockerfile that specifies the base image, copies the application code, installs dependencies, and defines the command to run the service.
* **Dependencies:**  
  All necessary dependencies are installed within the Docker containers, ensuring that the application runs smoothly without external dependencies.

**Benefits**

* **Consistency:**  
  Docker containers ensure that the application behaves identically across development, testing, and production environments.
* **Scalability:**  
  Services can be scaled horizontally by deploying multiple instances of containers as needed.

**Snapshots:**  
*Include screenshots of Dockerfile contents, docker-compose setup, and running Docker containers.*

**Validation and Sanitization**

*Contributors:* Saadallah Itani and Jad Ghamloush

Robust validation and sanitization mechanisms are implemented to secure the system against common vulnerabilities such as SQL injection and cross-site scripting (XSS).

**Implementation**

* **Input Validation:**
  + **Data Types:** Ensuring that inputs are of expected data types (e.g., integers for ratings).
  + **Value Ranges:** Enforcing constraints like ratings between 1 and 5, and non-negative stock counts.
* **Sanitization:**
  + **Parameterized Queries:** Utilizing parameterized SQL queries to prevent SQL injection attacks.
  + **Escaping Inputs:** Escaping or removing potentially malicious characters from user inputs.

**Security Measures**

* **Authentication Checks:**  
  Ensuring that only authenticated users can perform sensitive operations like submitting, updating, or deleting reviews.
* **Role-Based Access Control:**  
  Differentiating permissions based on user roles (user vs. admin) to restrict access to certain functionalities.

**Snapshots:**  
*Include code snippets demonstrating validation and sanitization logic, as well as examples of sanitized inputs.*

**User Authentication**

*Contributor:* Saadallah Itani

User authentication is a critical component of the Reviews Management API, ensuring that only authorized users can perform actions such as submitting, updating, or deleting reviews.

**Implementation**

* **Authentication Mechanism:**  
  Utilizes JSON Web Tokens (JWT) to manage user authentication and authorization.
* **Endpoints:**
  + POST /api/register: Register a new user with a username, password, and role.
  + POST /api/login: Authenticate a user and issue a JWT upon successful login.
* **Protected Routes:**  
  Routes that modify data, such as submitting or updating reviews, are protected using the @jwt\_required() decorator to ensure that only authenticated users can access them.
* **Role-Based Authorization:**  
  Custom decorators like admin\_required are implemented to restrict access to administrative functionalities, such as moderating reviews.

**Security Measures**

* **Password Hashing:**  
  User passwords are securely hashed using Werkzeug's security utilities before being stored in the database.
* **Token Security:**  
  JWTs are signed with a secret key (JWT\_SECRET\_KEY) to prevent tampering. Tokens include user roles to facilitate authorization checks.

**Snapshots:**  
*Include screenshots of successful registration and login processes, as well as examples of protected route access using JWT tokens.*

**Moderation**

*Contributor:* Jad Ghamloush

Moderation features are integral to maintaining the quality and reliability of reviews within the system. They allow for the management of inappropriate or flagged reviews.

**Implementation**

* **Flagging Reviews:**  
  Authenticated users can flag reviews that they deem inappropriate. This action sets the flagged status of a review to 1.
* **Moderating Reviews:**  
  Administrators have exclusive access to moderate flagged reviews. They can approve or reject reviews, which updates the moderated status accordingly:
  + **Approve:** Sets moderated to 1, indicating the review is validated.
  + **Reject:** Sets moderated to 0, indicating the review is invalidated.
* **Review Visibility:**  
  Only moderated reviews are visible to the public, ensuring that unapproved or inappropriate content is not displayed.

**Workflow**

1. **Flagging:**
   * User identifies an inappropriate review.
   * User sends a POST request to /api/reviews/flag/<review\_id> to flag the review.
2. **Moderation:**
   * Admin reviews flagged content.
   * Admin sends a POST request to /api/reviews/moderate/<review\_id> with the action (approve or reject).

**Challenges Faced**

* **Ensuring Timely Moderation:**  
  Implementing a system that allows for prompt review and action by administrators to maintain content quality.
* **Preventing Abuse:**  
  Implementing measures to prevent users from maliciously flagging reviews without valid reasons.

**Snapshots:**  
*Include screenshots of the flagging process, moderation actions, and the updated statuses of reviews.*

**References**

*Contributors:* Saadallah Itani and Jad Ghamloush

* **Flask Documentation:**  
  https://flask.palletsprojects.com/
* **SQLite Documentation:**  
  https://www.sqlite.org/docs.html
* **Flask-JWT-Extended Documentation:**  
  <https://flask-jwt-extended.readthedocs.io/>
* **Werkzeug Security Utilities:**  
  https://werkzeug.palletsprojects.com/en/2.0.x/utils/
* **Docker Documentation:**  
  https://docs.docker.com/
* **Pytest Documentation:**  
  https://docs.pytest.org/
* **Postman Documentation:**  
  https://learning.postman.com/docs/getting-started/introduction/

**Appendices**

**Appendix A: Code Snippets**

**Customers Service - Adding a Customer:**

python

Copy code

@app.route('/api/customers/add', methods=['POST'])

def api\_add\_customer():

customer = request.get\_json()

inserted\_customer = insert\_customer(customer)

if inserted\_customer:

return jsonify(inserted\_customer), 200

else:

return jsonify({}), 400

**Inventory Service - Deducting Stock:**

python

Copy code

@app.route('/api/goods/deduct/<int:good\_id>', methods=['PUT'])

def api\_deduct\_good(good\_id):

try:

data = request.get\_json()

if 'quantity' not in data:

return jsonify({"error": "Missing field: quantity"}), 400

quantity = data['quantity']

if not isinstance(quantity, int) or quantity <= 0:

return jsonify({"error": "Invalid quantity"}), 400

result = deduct\_good(good\_id, quantity)

if result["status"] == "Stock deducted successfully.":

return jsonify(result), 200

elif result["status"] == "Insufficient stock to deduct.":

return jsonify(result), 400

elif result["status"] == "Good not found.":

return jsonify(result), 404

else:

return jsonify(result), 500

except Exception as e:

return jsonify({"error": f"Invalid request: {e}"}), 400

**Reviews Service - Moderating a Review:**

python

Copy code

@app.route('/api/reviews/moderate/<int:review\_id>', methods=['POST'])

@admin\_required

def api\_moderate\_review(review\_id):

data = request.get\_json()

action = data.get('action')

if action not in ['approve', 'reject']:

return jsonify({"status": "Error: Invalid action. Use 'approve' or 'reject'."}), 400

# Fetch the review to verify it's flagged

try:

conn = connect\_to\_db()

cur = conn.cursor()

cur.execute("SELECT flagged FROM reviews WHERE review\_id = ?", (review\_id,))

review = cur.fetchone()

if not review:

return jsonify({"status": "Error: Review not found."}), 404

if not review['flagged']:

return jsonify({"status": "Error: Review is not flagged for moderation."}), 400

except Exception as e:

return jsonify({"status": f"Error: {str(e)}"}), 500

finally:

conn.close()

# Perform moderation

result = moderate\_review(review\_id, action)

return jsonify(result), 200

**Test Service1.py - Adding a Customer:**

def test\_add\_customer(client):

new\_customer = {

'full\_name': 'John Doe',

'username': 'johndoe',

'password': 'password123',

'age': 30,

'address': '123 Main St',

'gender': 'Male',

'marital\_status': 'Single'

}

response = client.post('/api/customers/add', json=new\_customer)

assert response.status\_code == 200, "Expected status code 200 for successful customer addition."

data = response.get\_json()

assert data['username'] == 'johndoe', "Returned username should be 'johndoe'."

*Additional code snippets can be included in this appendix as needed.*

**Note:** Replace placeholder links and paths with actual links relevant to your project. Ensure all snapshots and code snippets are properly captured and included in the final report document.

**Instructions to Convert to .docx:**

1. **Copy the Content:**
   * Select all the text above from the **Title Page** to the end of the **Appendices** section.
2. **Paste into Microsoft Word:**
   * Open Microsoft Word or any other word processing software.
   * Paste the copied content into a new document.
3. **Format the Document:**
   * **Headings:** Apply appropriate heading styles (e.g., Heading 1 for main sections, Heading 2 for subsections).
   * **Bold Text:** Ensure that bold texts (like section titles) are properly formatted.
   * **Code Snippets:** Use a monospaced font (like Courier New) and apply shading or borders to differentiate them from regular text.
   * **Snapshots:** Insert actual screenshots where placeholders are mentioned.
   * **Table of Contents:** Use Word's built-in Table of Contents feature to generate a dynamic TOC based on the heading styles.
4. **Save as .docx:**
   * Once formatting is complete, save the document in .docx format by selecting **File > Save As** and choosing the .docx option.
5. **Review the Document:**
   * Go through the document to ensure all sections are correctly formatted and all snapshots and code snippets are properly placed.

If you encounter any issues during the conversion process or need further assistance with specific sections, feel free to ask!