Final Project Design Document

Group 002

This project simulates an online furniture store, managing user accounts, inventory, shopping carts, order processing, and a RESTful API.

Design Modifications

Significant changes were made during development to improve performance, maintainability, and security.

Backend Architecture

- **Planned:** Serverless architecture using AWS Lambda.
- **Implemented:** Flask-based backend with a dedicated server.
- **Reasoning:**
  - Flask allows better debugging and session management.
  - AWS Lambda requires stateless interactions, making user management and order tracking difficult.

Database Choice

- **Planned:** PostgreSQL with Amazon RDS.
- **Implemented:** Pickle for inventory & orders, JSON for user storage.
- **Reasoning:**
  - Opted for file-based storage to simplify setup and reduce overhead.
  - **Pickle** retains Python object structures efficiently for inventory and orders.
  - **JSON** provides a structured, readable, and secure format for user data, facilitating integration with authentication mechanisms while minimizing security risks.

API Changes

- Switched to **session-based authentication** instead of token-based authentication.
- **Reasoning:**
  - Simplify login flow and improve user experience.
  - Reduces API overhead by eliminating frequent JWT decoding.
  - Allows easy session invalidation to enhance security.
  - Prevents token exposure vulnerabilities.

Model Changes

1. User Model (user.py)

- **Planned:**
  - User class with username, email, password, and address.
  - Subclasses: Client (with shopping cart), Manager (with role).
- **Implemented:**
  - Added bcrypt password hashing.
  - Implemented **UserDB Singleton** for centralized user management.
- **Reasoning:**
  - **Security:** Bcrypt prevents plaintext password storage.
  - **Maintainability:** Singleton pattern ensures a single source of truth.

2. Inventory Model (inventory.py)

- **Implemented:**
  - Switched to Pandas-based storage instead of SQL.
  - update_quantity() manually modifies stock levels.
- **Reasoning:**
  - Simplifies implementation and prevents redundant operations.
  - Pandas-based storage allows easier data handling.

3. Shopping Cart Model (cart.py)

- **Implemented:**
  - Matches UML but includes additional methods: validate_cart() and purchase().
  - Introduced **PaymentGateway** for future third-party payment integration.
- **Reasoning:**
  - validate_cart() prevents purchasing out-of-stock items.
  - PaymentGateway ensures adaptability for future enhancements.

4. Order Management (order.py)

- **Planned:** Create orders and manage history.
- **Implemented:**
  - **OrderManager** handles order history, payment processing, and inventory updates.
  - Support order cancellation and order status updates.

- **Reasoning:**
  - **Efficiency:** Pandas-based storage enables faster read performance.
  - **Simplification:** Pickle allows easy data retrieval and storage.

5. Factory Pattern (factory.py)

- **Planned:** Use a Factory Pattern for dynamic furniture creation.
- **Implemented:**
    - **FurnitureFactory** allows dynamic registration and creation of furniture types.
- **Reasoning:**
    - Ensures flexibility to add new furniture types without modifying core logic.
    - Simplify inventory management.

6. Authentication and Security (auth.py)

- **Planned:** JWT-based authentication.
- **Implemented:** Session-based authentication using Flask.
- **Reasoning:**
    - JWT is suited for distributed systems, but this project is monolithic.
    - Flask sessions provide built-in expiration management.
- **Implementation:**
    - authenticate_user() validates users.
    - require_auth and require_role to enforce access control.
    - Passwords are securely hashed using bcrypt.
    - Flask sessions expire after inactivity for added security.

Testing & Validation

- **test_auth.py** – Authentication and user validation.
- **test_inventory.py** – Inventory management functions.
- **test_cart.py** – Shopping cart operations.
- **test_order.py** – Order processing and history tracking.
- **test_factory.py** – FurnitureFactory dynamic object creation.
- **test_APIroutes.py** – API endpoint validation using mock requests.

The system was thoroughly tested to ensure reliability and correctness, utilizing regression testing to verify that future updates do not negatively impact existing functionality. Additionally, the testing process helped identify edge cases and prevent unintended failures, enhancing the system's stability and robustness.
A minimum code coverage (COV) of 80% was required, and the project successfully achieved 83% coverage, ensuring a high level of test completeness and system functionality.

Regression Test

The regression test we conducted aimed to validate the end-to-end functionality of our system, ensuring that core operations work seamlessly. It validates that all critical system functions operate correctly and remain stable after updates.

The Regression test followed a structured process:

1. System Setup: We initialized the system with test databases for users, inventory, and orders. Sample users and furniture items were added to simulate real-world scenarios.

2. Product Search & Cart Operations: We verified that products could be searched in the inventory and successfully added to a user's shopping cart.

3. Checkout Process: The test simulated a full purchase cycle, ensuring that payments were processed correctly, inventory was updated accordingly, and an order was recorded in the system.

4. Verification of Updates: Post-checkout, we validated that the purchased item quantities were correctly deducted from inventory and that the order manager reflected the new purchase.

5. Database Integrity Check: We confirmed that only the modified products were updated while ensuring that other inventory items remained unchanged.

6. Database Persistence & Cleanup: Finally, we ensured that updates were stored correctly in persistent files, mimicking real-world data storage.

Creativity and Additional Features in Our Online Store

1. Implementing an Abstract User Class to Support a Management Hierarchy

In our online store system, we designed an abstract User class to create a structured user hierarchy. This abstraction allows us to define different types of users with specific privileges while maintaining a common set of attributes and methods.

We introduced two distinct user roles:

- **Client**: A regular customer who can browse the store, add items to their shopping cart, and place orders.
- **Management**: A managerial user with enhanced privileges, such as:
  Adding new furniture items to the store, monitoring stock levels, and identifying low-inventory items.

This approach follows the principles of object-oriented programming (OOP) by ensuring reusability, scalability, and maintainability. The Management class extends the User class, inheriting core attributes while introducing managerial-specific functionalities.

By implementing this hierarchy, we ensure that business-critical actions, such as modifying inventory, are restricted to authorized users only.

2. Using Postman for API Testing

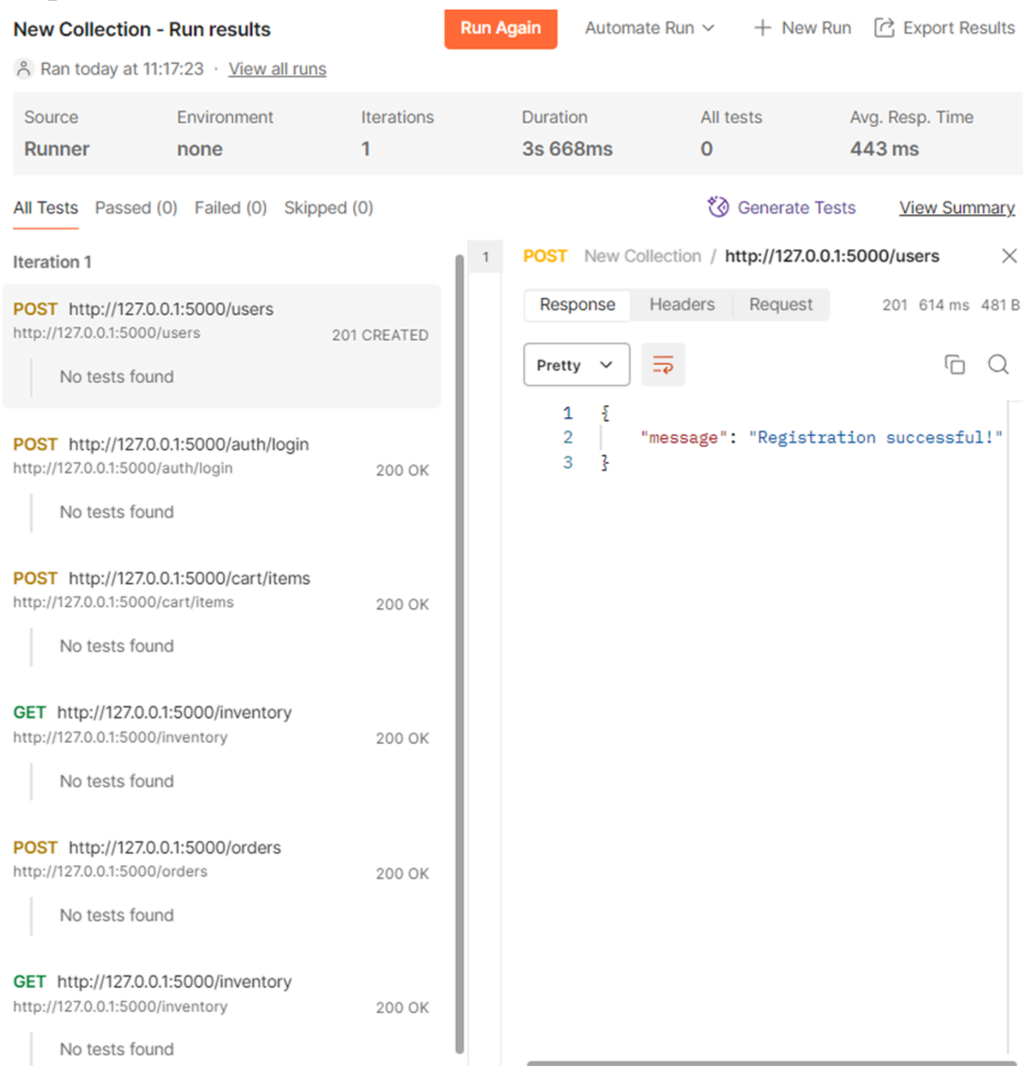
To ensure the reliability and correctness of our online store's API, we used Postman, a powerful tool for API development and testing. We created a full collection of API tests, allowing us to verify the behaviour of different endpoints. In this way, we make real case tests that allow us to check that all the implemented functionalities are working correctly and that all Databases are updating correctly and maintain consistency.

The use of Postman has several advantages. We made an automated test execution. We designed a collection of test cases that run sequentially (image below), ensuring that all API functionalities are validated in one go. In addition, Postman allowed us to test user registration, authentication, inventory management, and order placement systematically, which made a consistent testing process. We could inspect request responses, check HTTP status codes, and verify expected outcomes, making debugging easier.

By implementing a structured testing approach, we improved the reliability, security, and robustness of our API. This ensures a seamless experience for both regular customers and management users.
*** Note that Postman outputs are added on the next page.

These additional features enhance the functionality and maintainability of our online store, providing a well-structured user system and ensuring high-quality API performance through thorough testing.

Outputs from Postman test for API:



New Collection - Run results

Run Again · Automate Run · New Run · Export Results

Ran today at 11:17:23 · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	1	3s 668ms	0	443 ms

All Tests · Passed (0) · Failed (0) · Skipped (0) · Generate Tests · View Summary

Iteration 1

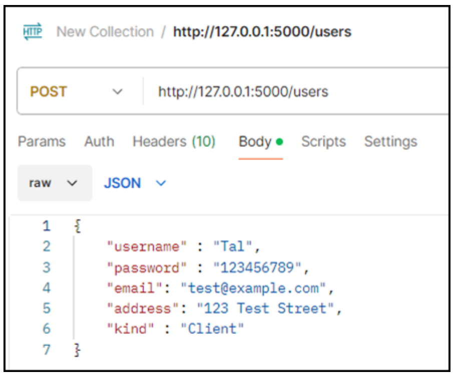
```
POST  http://127.0.0.1:5000/users
http://127.0.0.1:5000/users                201 CREATED
      No tests found

POST  http://127.0.0.1:5000/auth/login
http://127.0.0.1:5000/auth/login           200 OK
      No tests found

POST  http://127.0.0.1:5000/cart/items
http://127.0.0.1:5000/cart/items           200 OK
      No tests found

GET   http://127.0.0.1:5000/inventory
http://127.0.0.1:5000/inventory            200 OK
      No tests found

POST  http://127.0.0.1:5000/orders
http://127.0.0.1:5000/orders               200 OK
      No tests found

GET   http://127.0.0.1:5000/inventory
http://127.0.0.1:5000/inventory            200 OK
      No tests found
```

POST New Collection / http://127.0.0.1:5000/users

Response · Headers · Request — 201 614 ms 481 B

Pretty

```
{
    "message": "Registration successful!"
}
```



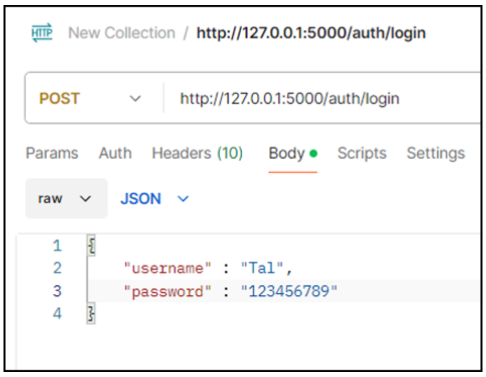
New Collection / http://127.0.0.1:5000/users

POST http://127.0.0.1:5000/users

Params · Auth · Headers (10) · Body · Scripts · Settings

raw · JSON

```
{
    "username" : "Tal",
    "password" : "123456789",
    "email": "test@example.com",
    "address": "123 Test Street",
    "kind" : "Client"
}
```



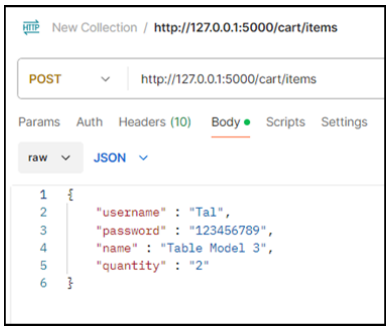
New Collection / http://127.0.0.1:5000/auth/login

POST http://127.0.0.1:5000/auth/login

Params · Auth · Headers (10) · Body · Scripts · Settings

raw · JSON

```
{
    "username" : "Tal",
    "password" : "123456789"
}
```



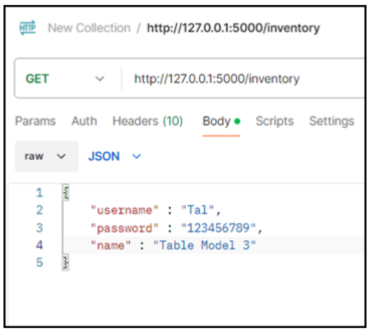
New Collection / http://127.0.0.1:5000/cart/items

POST http://127.0.0.1:5000/cart/items

Params · Auth · Headers (10) · Body · Scripts · Settings

raw · JSON

```
{
    "username" : "Tal",
    "password" : "123456789",
    "name" : "Table Model 3",
    "quantity" : "2"
}
```



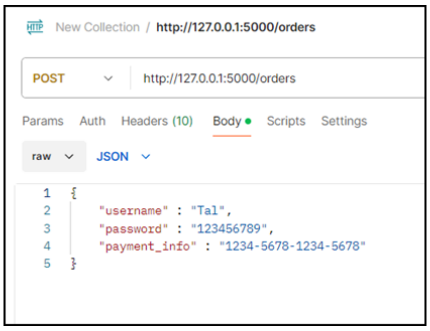
New Collection / http://127.0.0.1:5000/inventory

GET http://127.0.0.1:5000/inventory

Params · Auth · Headers (10) · Body · Scripts · Settings

raw · JSON

```
{
    "username" : "Tal",
    "password" : "123456789",
    "name" : "Table Model 3"
}
```



New Collection / http://127.0.0.1:5000/orders

POST http://127.0.0.1:5000/orders

Params · Auth · Headers (10) · Body · Scripts · Settings

raw · JSON

```
{
    "username" : "Tal",
    "password" : "123456789",
    "payment_info" : "1234-5678-1234-5678"
}
```