Technical Report: ecommerce_books

Author: Jonathan Castellanos

August 8, 2025

Contents

1 Introduction

This document presents the technical report for the development of the project **ecommerce_books**, a web-based bookstore application. The goal of the application is to serve a small business with an estimated traffic of 100 users per day. This first version aims to register and log in customers, manage product listings, inventory control, and sales operations.

2 Objectives

The primary objectives of this project include:

- Provide a user-friendly web interface for book shopping.
- Implement a secure authentication mechanism for users.
- Allow inventory and product management.
- Create a sales register.
- Improve sales volume with solution.

3 Scope

The scope of this project was limited to the design and implementation of the fundamental modules for an online bookstore application (**ecommerce_books**). This includes:

- User registration and authentication
- Product management (add books)
- Inventory control
- Sales register
- Basic interfaces

The application was built as a functional prototype to demonstrate core functionalities necessary for a minimal viable product (MVP).

The project **excludes** advanced features such as:

- Payment gateway integration
- User reviews or recommendations
- Order tracking
- Full e-commerce analytics

This focus helped to concentrate development efforts on building a reliable base infrastructure.

4 Assumptions

The following assumptions were made during the planning and development of this project:

- The application is intended to serve a small business with max 100 users per day.
- The system will be deployed in a controlled environment with limited traffic spikes.
- Only one administrator manages the backend operations.

- Data used for testing represents realistic scenarios.
- Security mechanisms are limited to JWT-based authentication for the MVP stage.
- The system is expected to run on a single server instance.

These assumptions influenced the architectural and technological choices made during the first development phase.

5 Limitations

This project faced several limitations, primarily due to resource constraints:

- **Team size:** The project was developed entirely by a single developer, which constrained the implementation timeline and scope.
- **Time:** The available time allowed only the development of essential modules for the MVP. Features such as scalability optimization are planned for later phases.
- **UI/UX:** Frontend design was kept minimal and functional, with limited focus on user experience aesthetics.
- **External integrations:** No third-party payment or shipping services were included in this version.

Despite these limitations, the core business logic and application structure were implemented successfully, providing a strong foundation for future improvements.

6 Methodology

This project followed a agile methodology, which included the following key phases:

6.1 Visual Story Mapping

To visualize the product journey, a Story Map was created to group features by user activity for create user hisroty. (Figure 1: Visual Story Map)

6.2 Requirements and User Stories

Requirements were collected based on the needs of a small business looking to digitalize its book sales. The core user stories defined were:

Code	HU-001
Priority	Medium
Story Points	3
User Story	As an administrator, I want to create a new product by entering its information, so that it can be available in the system.
Acceptance Criteria	• When the user enters the name, description, price, and initial quantity, then the product is saved in the database. • When the product is created, the system responds with the product information.
Alternative Flow	• Given that the information is incomplete or invalid, then the system returns an error. • When a database error occurs while saving the product, then the system returns a 500 error.

Table 1: User Story HU-001 – Create Product

Code	HU-002
Priority	Medium
Story Points	3
User Story	As an administrator, I want to update the inventory of a product so that the available quantity reflects the actual stock.
Acceptance Criteria	• Given the user sends a quantity for an existing product, then the system updates the inventory accordingly. • When the inventory is updated, the system responds with the new stock.
Alternative Flow	• Given the user updates the information and the product does not exist, then the system returns an error. • Given the user updates the information and the quantity is not a positive number, then the system returns an error. • When a database error occurs while updating the inventory, then the system returns a 500 error.

Table 2: User Story HU-002 – Update Inventory

Code	HU-003
Priority	Medium
Story Points	5
User Story	As a logged-in user, I want to purchase products from the system so that they can be shipped to my address.
Acceptance Criteria	• Given that the user is authenticated, selects an existing product, and provides a quantity, the system validates the stock and deducts the correct amount. • Given a purchase is made and the user has a shipping address, then a shipment is registered.
Alternative Flow	• If the user is not authenticated, then the system returns an error. • If there is insufficient stock, then the system returns an error. • If a database error occurs while updating inventory or registering the shipment, then the system returns a 500 error.

Table 3: User Story HU-003 – Purchase Products

Code	HU-004
Priority	High
Story Points	3
User Story	As a new user, I want to register in the system so that I can make purchases.
Acceptance Criteria	• Given the user fills the form with a valid name, unique email (with '@' and '.'), valid password (minimum 8 alphanumeric characters), and valid shipping address, then the system registers the user.
Alternative Flow	• If the email is already registered, then the system returns an error. • If the password does not meet the criteria, then the system returns an error. • If a database error occurs while saving the user, then the system returns a 500 error.

Table 4: User Story HU-004 – User Registration

Code	HU-005
Priority	High
Story Points	3
User Story	As a registered user, I want to log into the system so that I can access it.
Acceptance Criteria	• Given the user fills the login form with a valid email and password, then the system authenticates the user.
Alternative Flow	• If the credentials are incorrect, then the system returns an error. • If the user fails to log in after 3 attempts, then the system locks the account. • If a database error occurs during login, then the system returns a 500 error.

Table 5: User Story HU-005 – User Login

6.3 Business Process Modeling (BPMN)

A BPMN diagram was developed to describe the four main workflows. Register/log in, purchase products, create products, and update inventory. This was primarily used to identify the components in which each role will act. (Figure 2: BPMN Diagram)

6.4 Causal Loop Diagram

A causal loop diagram was created to identify feedback loops within the system, such as the relationship between book sales, software investment, and user ratings. This was done to understand which needs our service must address in order to improve and grow the business.(Figure 3: Casual Loop Diagram)

User rating features are expected to be implemented in the future.

6.5 Stock and Flow Diagram

The flowchart and stock diagram models the dynamics of inventory, specifically how stock is added and removed through user purchases.(Figure 4: Stock And Flow Diagram)

This allows us to understand how the system works and optimize inventory management in future implementations.

6.6 Architecture Design

The application's conceptual design was guided by domain-oriented principles. A layered architecture was used, dividing the system into four main layers: Presentation, Service, Business Logic, and Data. This decision was made based on the non-functional requirements of the initial implementation, which considered a low user base, in addition to implementing modules that address basic system functionality. The choice of this architecture also allowed for reduced development time and costs, considering staffing limitations.(Figure 5: Architecture Diagram)

6.7 Class Diagram

The class diagram defines the main entities and their relationships. (Figure 6: Class Diagram)

6.8 Data Modeling

The following entities were identified for the initial implementation of the service. (Figure 7: Entity-relation Diagram)

For the design of this solution, a relational database was chosen due to its ability to represent the domain's key entities in a structured and consistent manner. Relationships require high integrity; therefore, an SQL database allows us to easily address this. Finally, the database supports ACID transactions, which are vital for maintaining data consistency, especially in inventory and sales management. (Figure 8: Database Diagram)

7 Results

- A fully functional prototype of the ecommerce application was developed.
- Integration tests confirmed the correctness of:
  - User creation and login flows
  - Book creation and inventory management
  - Sales recording endpoints
- Full application integration (Frontend, Backend, Database)

Image basic view (Figure 9: First look layout)

8 Unit Test

8.1 Test Design

Table 6: Unit Test Design for InventoryBusiness Class

Test ID	**Test Objective**	**Test Input**	**Expected Output**
TC01	Verify successful inventory update when valid data is provided	‘{ ”idProduct”: 1, ”quantity”: 5 }‘	Returns the same data after mock update is called
TC02	Verify behavior when ‘idProduct‘ is missing from input data	‘{ ”quantity”: 5 }‘	Raises ‘ValueError‘: ”The field ’idProduct’ is required.”
TC03	Verify behavior when quantity is negative	‘{ ”idProduct”: 1, ”quantity”: -3 }‘	Raises ‘ValueError‘: ”The ’quantity’ must be a non-negative integer.”
TC04	Simulate database error during inventory update	‘{ ”idProduct”: 1, ”quantity”: 5 }‘ with DB mock raising exception	Raises ‘Exception‘: ”DB error”
TC05	Verify successful retrieval of inventory by product ID	‘product id = 1‘ with mock returning data	Returns ‘{ ”idProduct”: 1, ”quantity”: 20 }‘
TC06	Validate behavior with invalid product ID (negative)	‘product id = -10‘	Raises ‘ValueError‘: ”The ’product id’ is not valid”
TC07	Simulate database error during inventory fetch	‘product id = 1‘ with mock raising exception	Raises ‘Exception‘: ”DB error”

Table 7: Unit Test Design for ProductBusiness Class

Test ID	Test Objective	Test Input	Expected Output
TC01	Verify successful product creation with valid data	{ "name": "Laptop", "price": 1200.0, "description": "Gaming laptop" }	Returns the same data and calls the create function once
TC02	Validate behavior when 'name' is missing	{ "price": 100.0, "description": "Item" }	Raises 'ValueError': "The field 'name' is required."
TC03	Validate behavior when 'price' is not a number	{ "name": "Pen", "price": "cheap", "description": "Blue ink" }	Raises 'ValueError': "The field 'price' must be a number"
TC04	Simulate a persistence error during product creation	{ "name": "TV", "price": 500.0, "description": "Smart TV" } with mock raising exception	Raises 'Exception': "DB Error"
TC05	Verify successful retrieval of product by ID	'product id = 1' with mocked return of a product	Returns { "id": 1, "name": "Mouse", "price": 25.0, "description": "Wireless" }
TC06	Validate behavior when product ID is invalid (negative)	'product id = -9'	Raises 'ValueError': "Product ID is not valid."
TC07	Validate behavior when product not found by ID	'product id = 999' with mock returning 'None'	Raises 'ValueError': "Product not found."
TC08	Simulate persistence error during get by ID	'product id = 1' with mock raising exception	Raises 'Exception': "DB Failure"
TC09	Verify successful retrieval of all products	No input; mock returns list of products	Returns { { "id": 1, "name": "Keyboard" }, { "id": 2, "name": "Monitor" } }
TC10	Simulate persistence error during retrieval of all products	No input; mock raises exception	Raises 'Exception': "Read error"

Table 8: Unit Test Design for PurchaseService Class

Test ID	Test Objective	Test Input	Expected Output
TC01	Verify successful product purchase with all valid inputs	{ "idProduct": 1, "idUser": 2, "quantity": 3 }, with mocks for product, inventory, update, and create methods	Returns a dict with '"status": "AC"' and '"total": 300.0'
TC02	Validate behavior when 'idProduct' is missing	{ "idUser": 2, "quantity": 3 }	Returns a dict with '"error": "'idProduct' is required."'
TC03	Validate behavior when 'quantity' is not an integer	{ "idProduct": 1, "idUser": 2, "quantity": "five" }	Returns a dict with '"error": "'quantity' must be of type ¡class 'int'¿."'
TC04	Validate behavior when 'quantity' is less than 1	{ "idProduct": 1, "idUser": 2, "quantity": 0 }	Raises 'ValueError': "The 'quantity' must be a non-negative integer."

Test ID	Test Objective	Test Input	Expected Output
TC05	Simulate missing product in database	{ "idProduct": 1, "idUser": 2, "quantity": 3 } with mock raising 'ValueError("Product not found.")'	Raises 'ValueError': "Product not found."
TC06	Validate behavior when inventory is insufficient	{ "idProduct": 1, "idUser": 2, "quantity": 10 } with product and inventory mocked; inventory quantity = 5	Raises 'ValueError': "Insufficient inventory quantity for this purchase."
TC07	Simulate persistence failure during purchase creation	{ "idProduct": 1, "idUser": 2, "quantity": 2 } with mocks and 'create purchase' raising exception	Raises 'Exception': "DB failure"

Table 9: Unit Test Design for UserBusiness Class

Test ID	Test Objective	Test Input	Expected Output
TC01	Verify successful user registration with valid data	{ "username": "john", "email": "john@example.com", "password": "securepass", "direction": "123 Main St" }	Returns { "message": "User successfully registered" } and calls 'create user' once
TC02	Validate behavior when 'username' is missing during registration	{ "email": "john@example.com", "password": "securepass", "direction": "123 Main St" }	Raises 'ValueError': "Missing field: username"
TC03	Validate behavior when email is already registered	{ "username": "john", "email": "john@example.com", "password": "securepass", "direction": "123 Main St" } with 'get user by email' returning a user	Raises 'ValueError': "Email is already registered"
TC04	Simulate persistence error during registration	Same input as TC01 with 'create user' raising 'Exception("DB Error")'	Raises 'Exception': "DB Error"
TC05	Verify successful user login with correct credentials	Valid 'username' and 'password', user found and password matches	Returns a dictionary containing a 'token'
TC06	Validate behavior when 'username' is missing during login	{ "password": "pass123" }	Raises 'ValueError': "Missing field: username"
TC07	Validate login with non-existent username	{ "username": "unknown", "password": "somepass" } with 'get user by username' returning 'None'	Raises 'ValueError': "Invalid username or password"
TC08	Validate login with incorrect password	Valid 'username' with hashed password mismatch	Raises 'ValueError': "Invalid username or password"
TC09	Simulate persistence error during login	Valid input with 'get user by username' raising 'Exception("DB Fail")'	Raises 'Exception': "DB Fail"

8.2 Test execution

- Inventory test (Figure 10: Inventory test)
- Product test (Figure 11: Product test)
- Purchase test (Figure 12: Purchase test)
- User test (Figure 13: User test)

9 Integration Test

9.1 Test Design

Table 10: Integration Test Design from Postman Collection

Test ID	Test Objective	Test Input	Expected Output
TI01	Verify the backend responds to a basic GET request at root endpoint	Method: GET URL: {{baseURL}}/	200 OK, response message such as "Hello World"
TI02	Test user registration with valid credentials	Method: POST URL: {{baseURL}}/api/users/register Body: { "username": "admin", "password": "admin", "email": "admin@gmail.com", "direction": "Cra 1 # 2 - 3" }	201 Created, message "User successfully registered"
TI03	Validate behavior when 'email' is missing during register	Method: POST URL: {{baseURL}}/api/users/register Body: { "username": "admin", "password": "admin", "direction": "Cra 1 # 2 - 3" }	400 Created, message "detail": "Missing field: email"
TI04	Validate behavior when 'email' registered	Method: POST URL: {{baseURL}}/api/users/register Body: { "username": "admin2", "password": "admin2", "email": "admin@gmail.com", "direction": "Cra 1 # 2 - 3" }	400 Created, message "Email is already registered"
TI05	Test login with correct credentials	Method: POST URL: {{baseURL}}/api/users/login Body: { "username": "admin", "password": "admin" }	200 Ok, { "token": "eyJh....." }

Test ID	Test Objective	Test Input	Expected Output
TI06	Validate behavior when ‘password‘ is missing during login	Method: POST URL: {{baseURL}}/api/users/login Body: { "username": "admin" }	400 Bad Request, message ”Missing field: password”
TI07	Test login with incorrect password	Method: POST URL: {{baseURL}}/api/users/login Body: { "username": "admin", "password": "adminBad" }	401 Unauthorized, error ”Invalid username or password”
Ti08	Test product creation with valid data	Method: POST URL: {{baseURL}}/api/products/ Body: { "name": "Product 3", "price": 50000.50, "description": "Description 3" }	201 Created, returns created product with ID
Ti09	Validate behavior when ‘description‘ is missing during create product	Method: POST URL: {{baseURL}}/api/products/ Body: { "name": "Product 3", "price": 50000.50 }	400 Bad request, message ”The field ’description’ is required”
TI10	Test updating inventory with valid values	Method: PUT URL: {{baseURL}}/api/inventory /update Body: { "idProduct": 1, "quantity": 50 }	200 OK, return { "idProduct": 1, "quantity": 50 }
TI11	Validate behavior when ‘quantity‘ is missing during update inventory	Method: PUT URL: {{baseURL}}/api/inventory /update Body: { "idProduct": 1 }	400 Bad Request, message ”The field ’quantity’ is required”
TI12	Validate behavior when ‘quantity‘ have bad value during update inventory	Method: PUT URL: {{baseURL}}/api/inventory /update Body: { "idProduct": 1, "quantity": -5 }	400 Bad Request, message ”The ’quantity’ must be a non-negative integer”
TI13	Test purchasing a product with valid data	Method: POST URL: {{baseURL}}/api/purchase/ Body: { "idProduct": 1, "idUser": 1, "quantity": 5 }	200 OK, return { "idProduct": 1, "idUser": 1, "quantity": 5, "total": 50000, "status": "AC" }
TI14	Validate behavior when ‘quantity‘ is missing during purchase	Method: POST URL: {{baseURL}}/api/purchase/ Body: { "idProduct": 1, "idUser": 1 }	400 Bad Request, message ”The ’quantity’ is required”

Test ID	Test Objective	Test Input	Expected Output
TI15	Validate behavior when ‘quantity‘ have bad value during purchase	Method: POST URL: {{baseURL}}/api/purchase/ Body: { "idProduct": 1, "idUser": 1, "quantity": -6 }	400 Bad Request, message ”The ’quantity’ must be a non-negative integer”
TI16	Validate behavior when inventory is insufficient during purchase	Method: POST URL: {{baseURL}}/api/purchase/ Body: { "idProduct": 1, "idUser": 1, "quantity": 100 }	400 Bad Request, message ”Insufficient inventory quantity for this purchase”
TI17	Retrieve all products from catalog	Method: GET URL: {{baseURL}}/api/products/	200 OK, returns a list of available products

9.2 Test execution

For integration tests, we opted to use API Test. The tests are located in the Postman collection within the project. (Figure 14: TI postman execution)

10 System Test

10.1 Test Design

Table 11: Stress Test Design

Test ID	Test Objective	Test Input	Expected Output
ST01	Evaluate system behavior under 100 concurrent purchase operations	100 users performing authenticated POST requests to ‘/purchase‘, each with valid product ID and quantity of 1–3	Response time remains under 2 seconds for 95% of requests; error rate below 2%
ST02	Evaluate login stability under high concurrency	300 users sending POST requests to ‘/login‘, using both valid and invalid credentials	At least 95% of valid logins succeed with response time ¡ 1.5s; invalid logins return correct 401 error; no system crash
ST03	Stress test on products view	100 users simultaneously requesting products via GET to ‘/products‘	System handles concurrent requests without timeouts or memory overload; 90% of responses returned within 3 seconds

10.2 Test execution

JMeter was used to run the stress tests. All three tests were completed successfully, responding to all requests with correct statuses within the established timeframes. (Figure 15: System test)

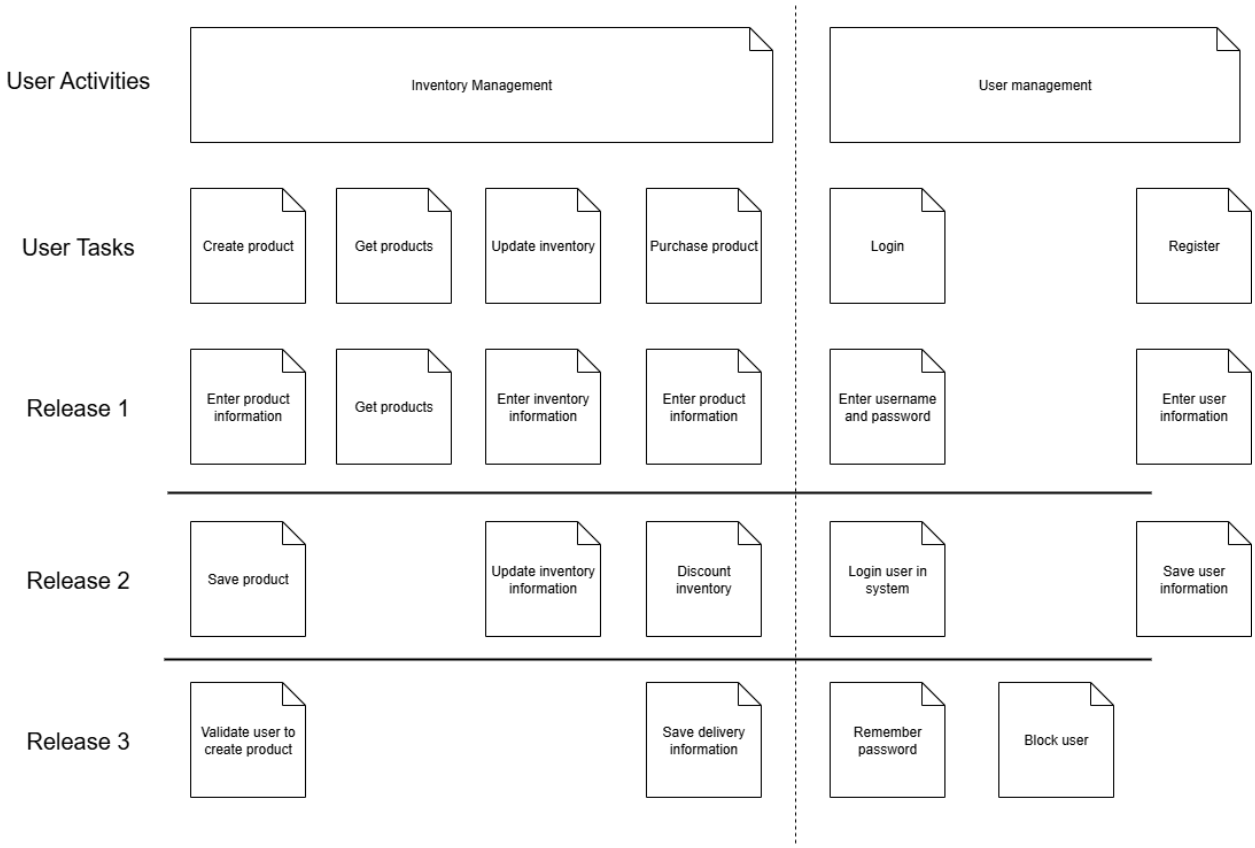

Figure 1: Visual Story Map

11 Discussion

The results obtained from the integration test collection demonstrate a comprehensive and structured approach to evaluating API functionality and reliability. The Postman collection organizes requests into well-defined folders representing different API modules, facilitating modular testing and ease of use.

The tests cover a range of scenarios, including successful requests, error handling, and edge cases. This indicates a correct validation of software.

Overall, the implementation met all expectations for this first phase. However, there are areas where improvements could be made. Several features were mentioned in the design that should be implemented in the future to meet all business needs.

12 Conclusion

In conclusion, the analyzed tests represent a well-designed and coded solution. Key findings indicate that the test suite effectively covers functional requirements and error conditions. It provides a reliable framework for validating system behavior in various scenarios. The clear organization of requests and the automation of test validations contribute to the efficiency and consistency of the testing process.

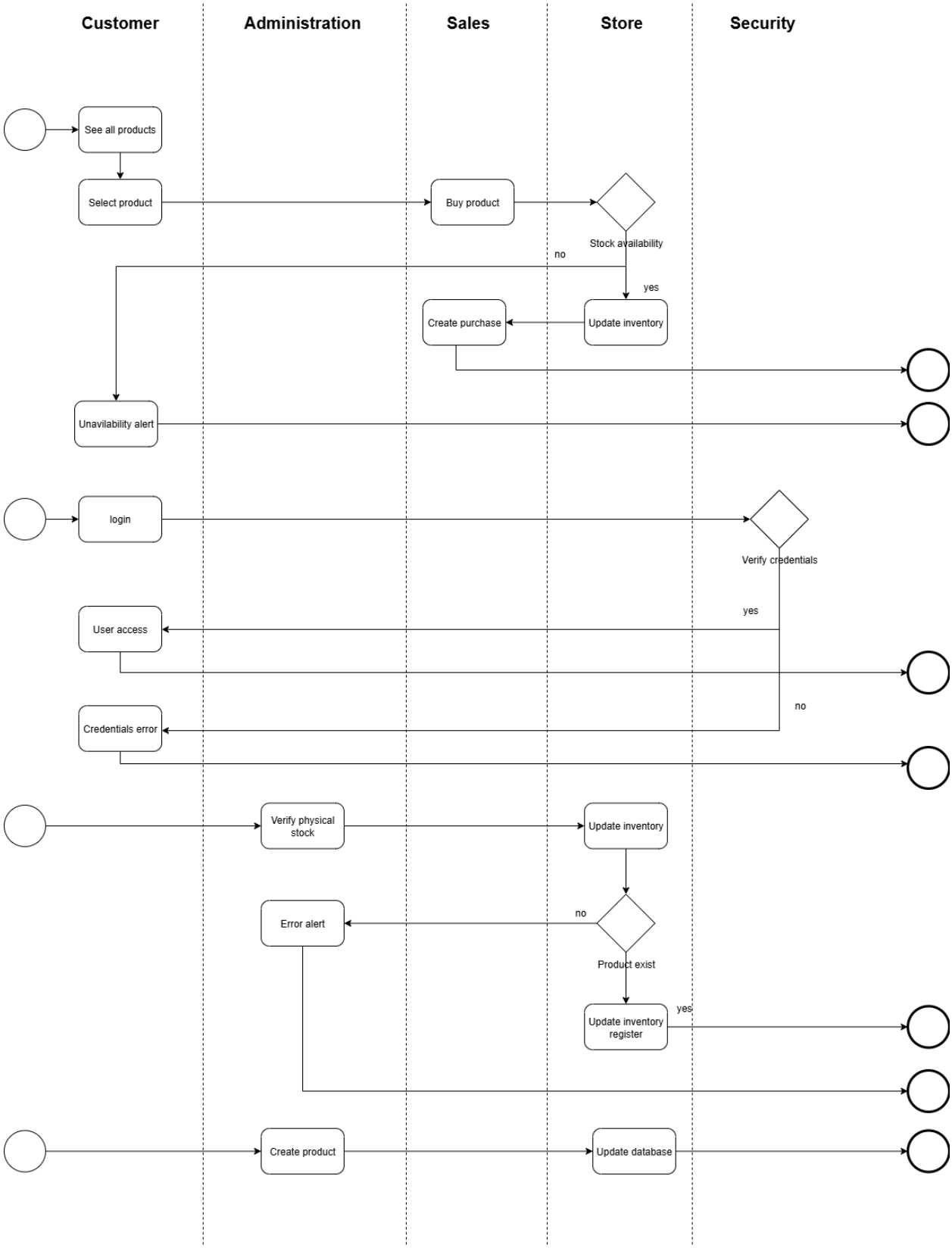

Figure 2: BPMN Diagram

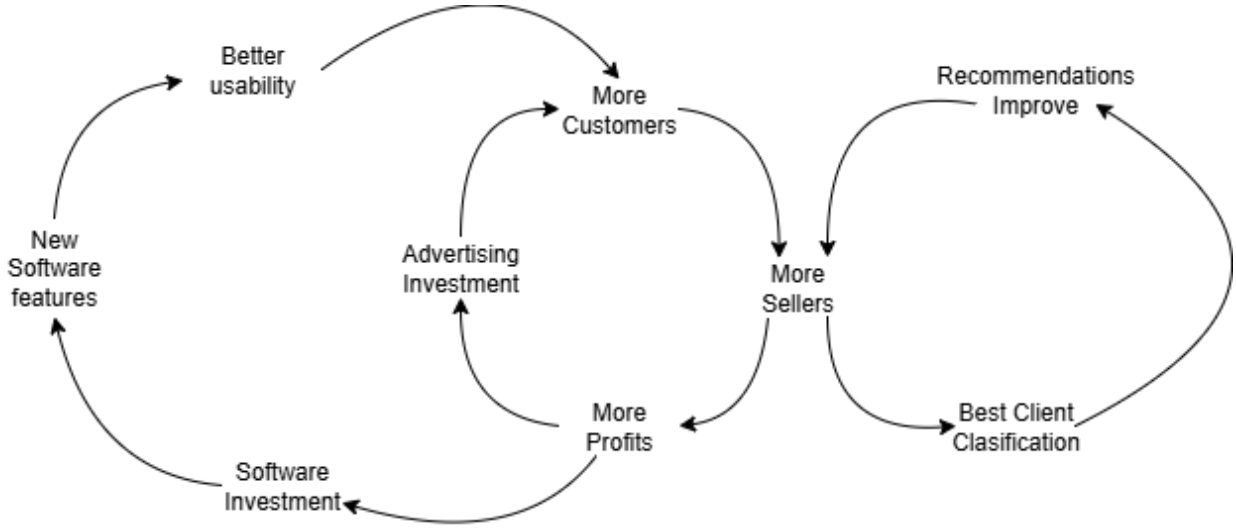

Figure 3: Casual Loop Diagram

13 Images

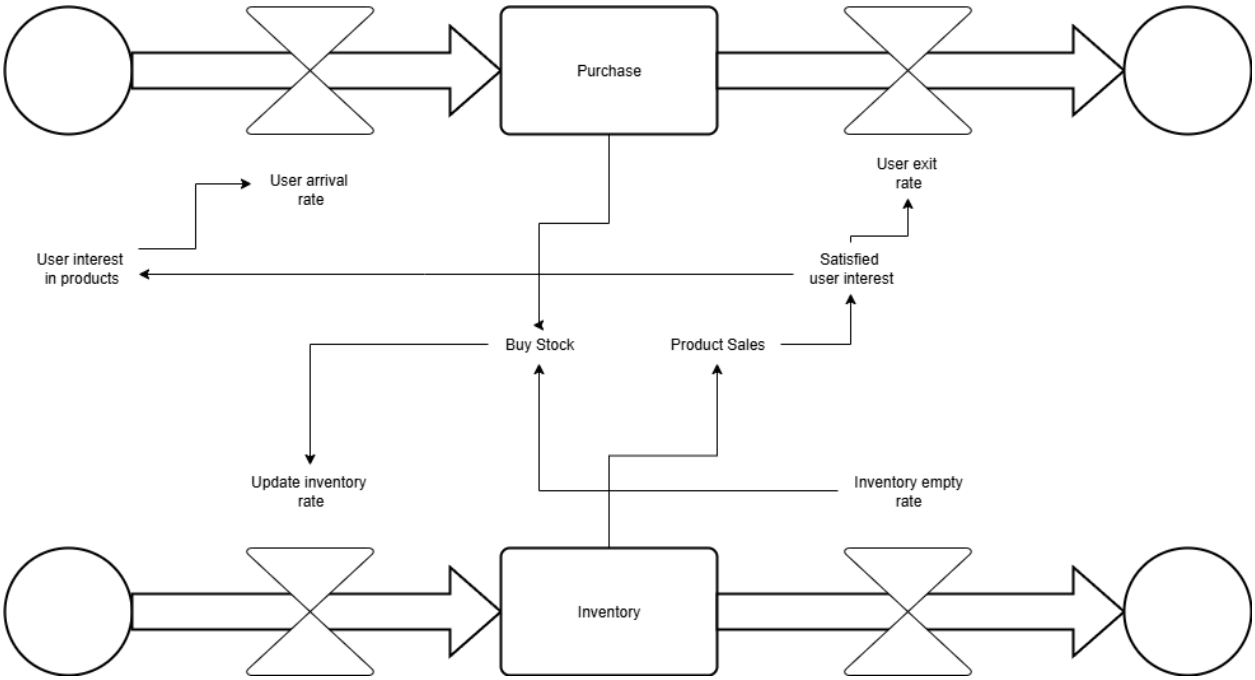

Figure 4: Stock And Flow Diagram

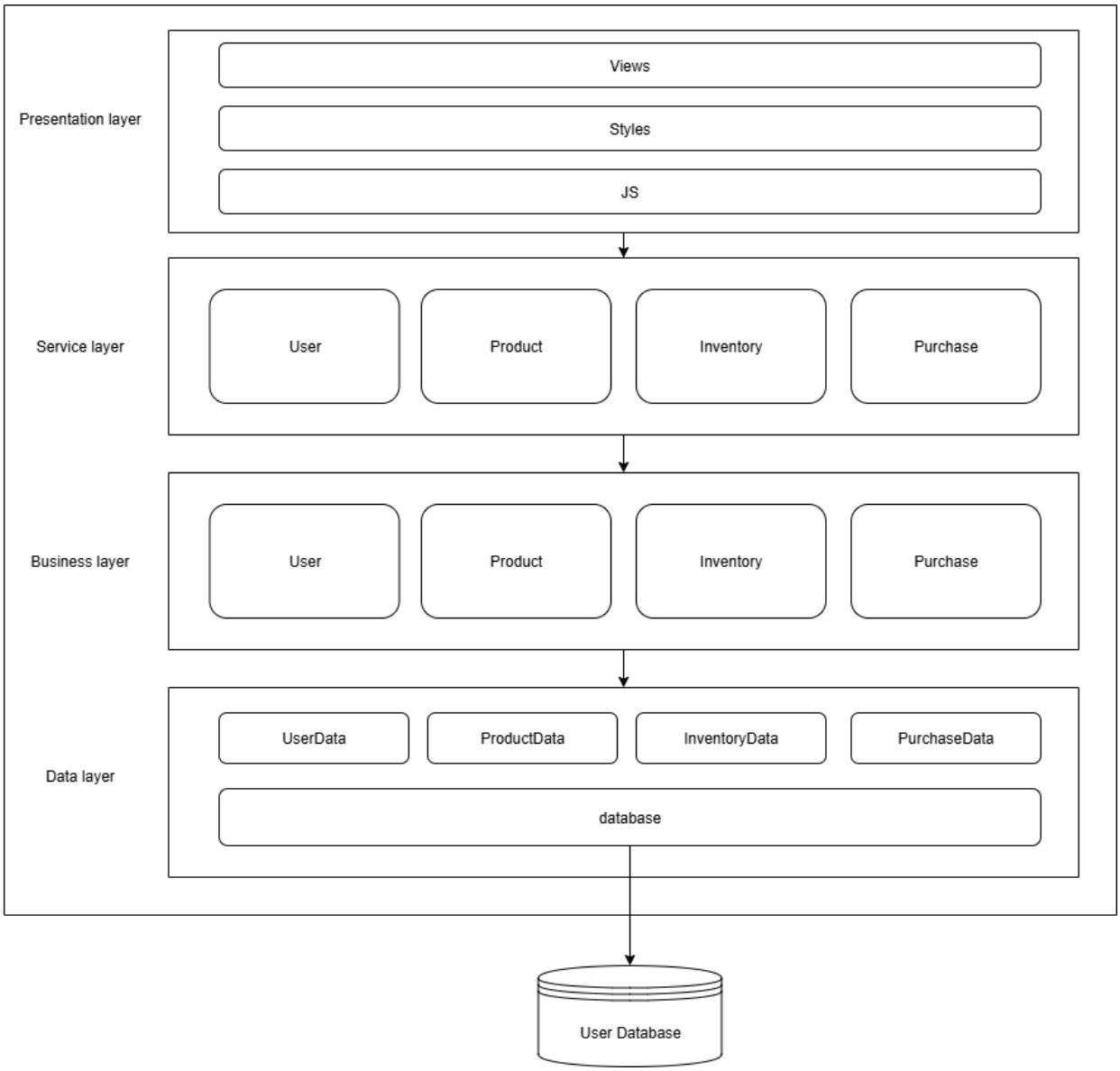

Figure 5: Architecture Diagram

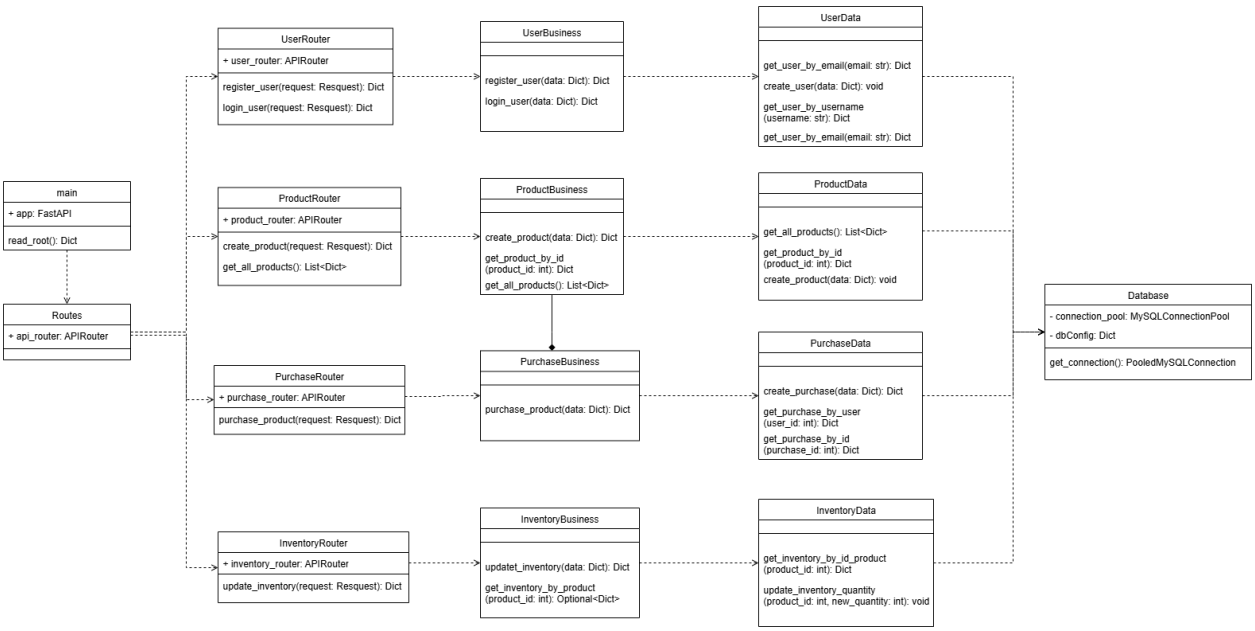

Figure 6: Class diagram

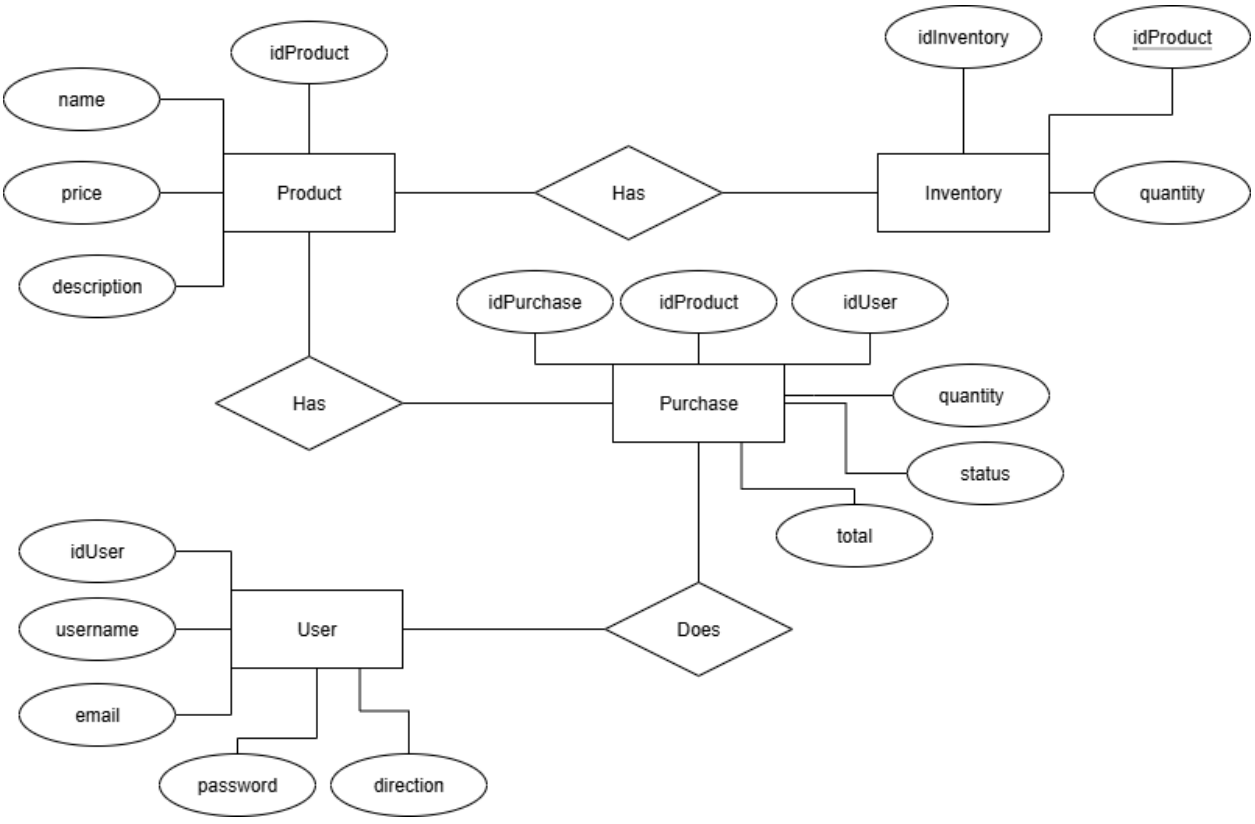

Figure 7: Entity-relation Diagram

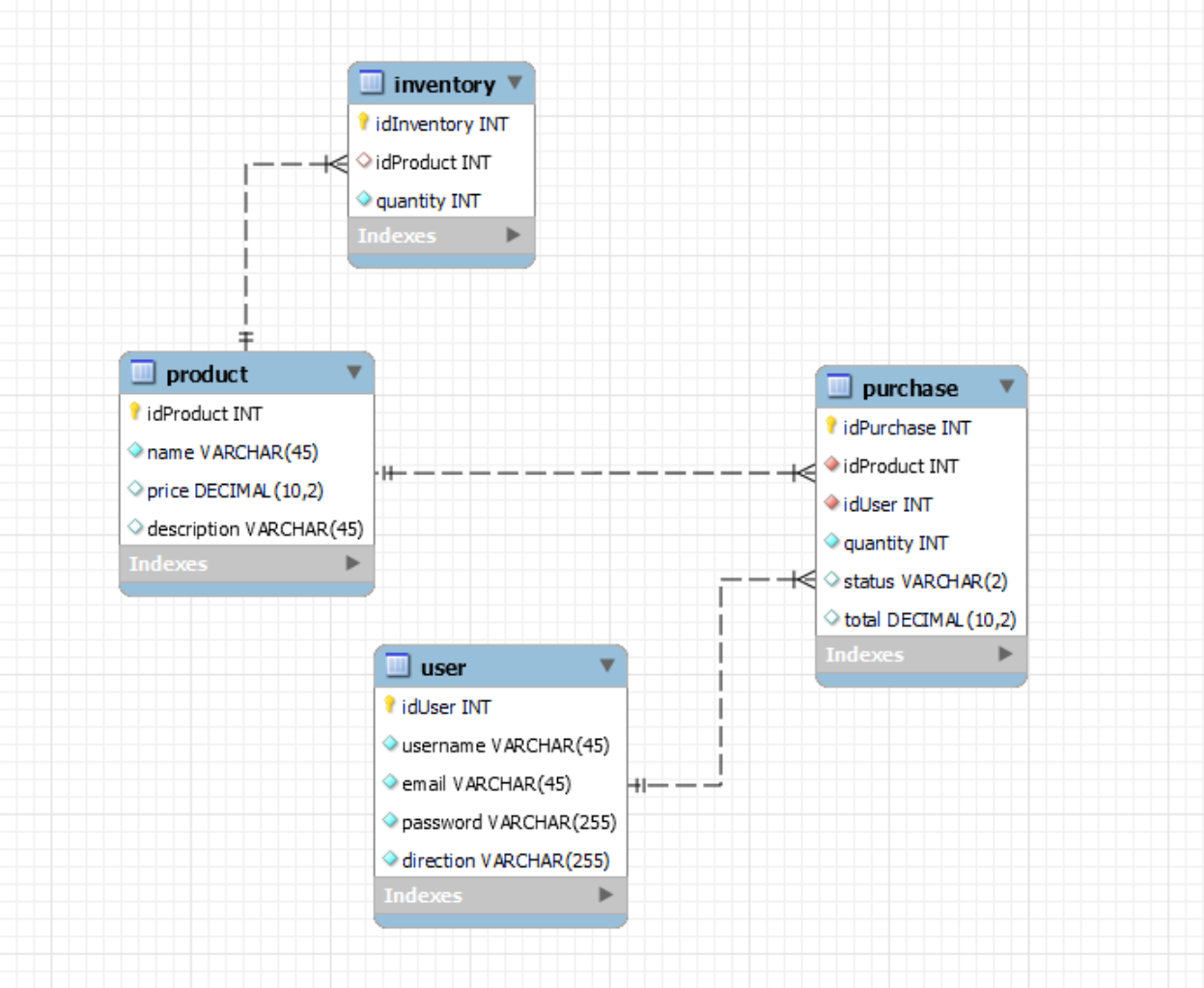

Figure 8: Database Diagram

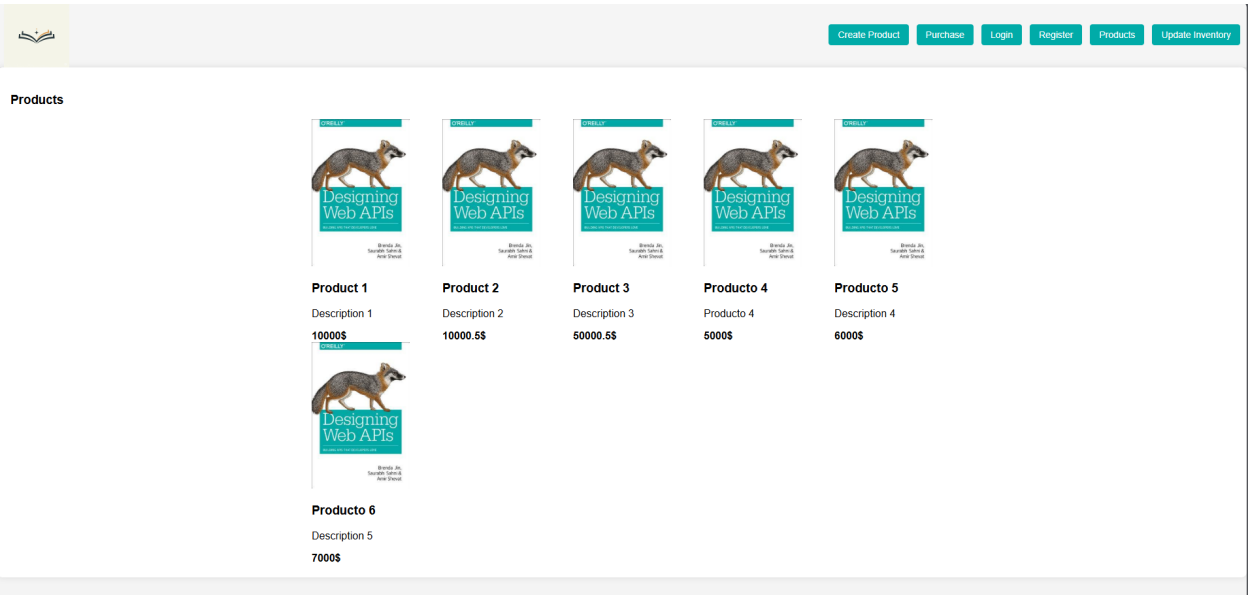

Figure 9: First look layout

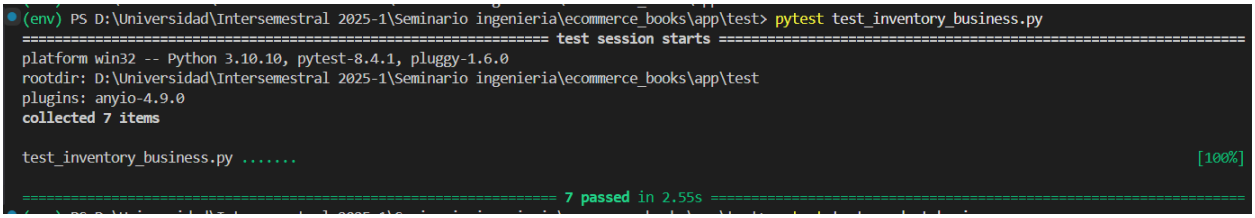

Figure 10: Inventory test

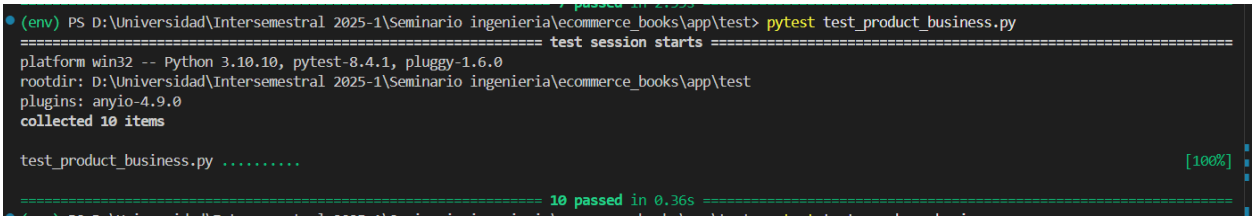

Figure 11: Product test

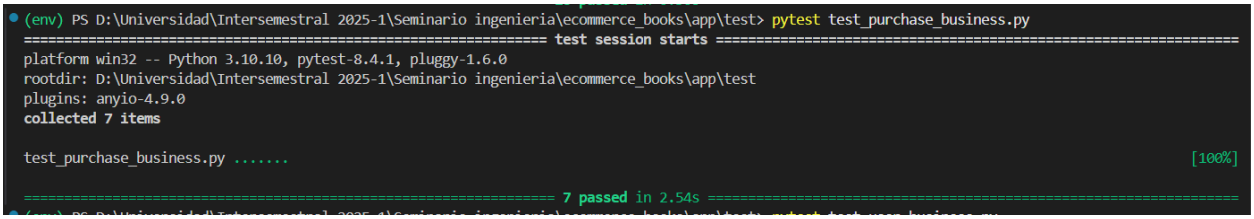

Figure 12: Purchase test

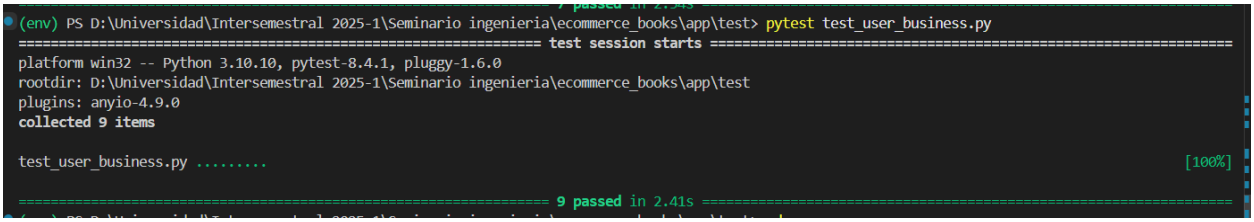

Figure 13: User test

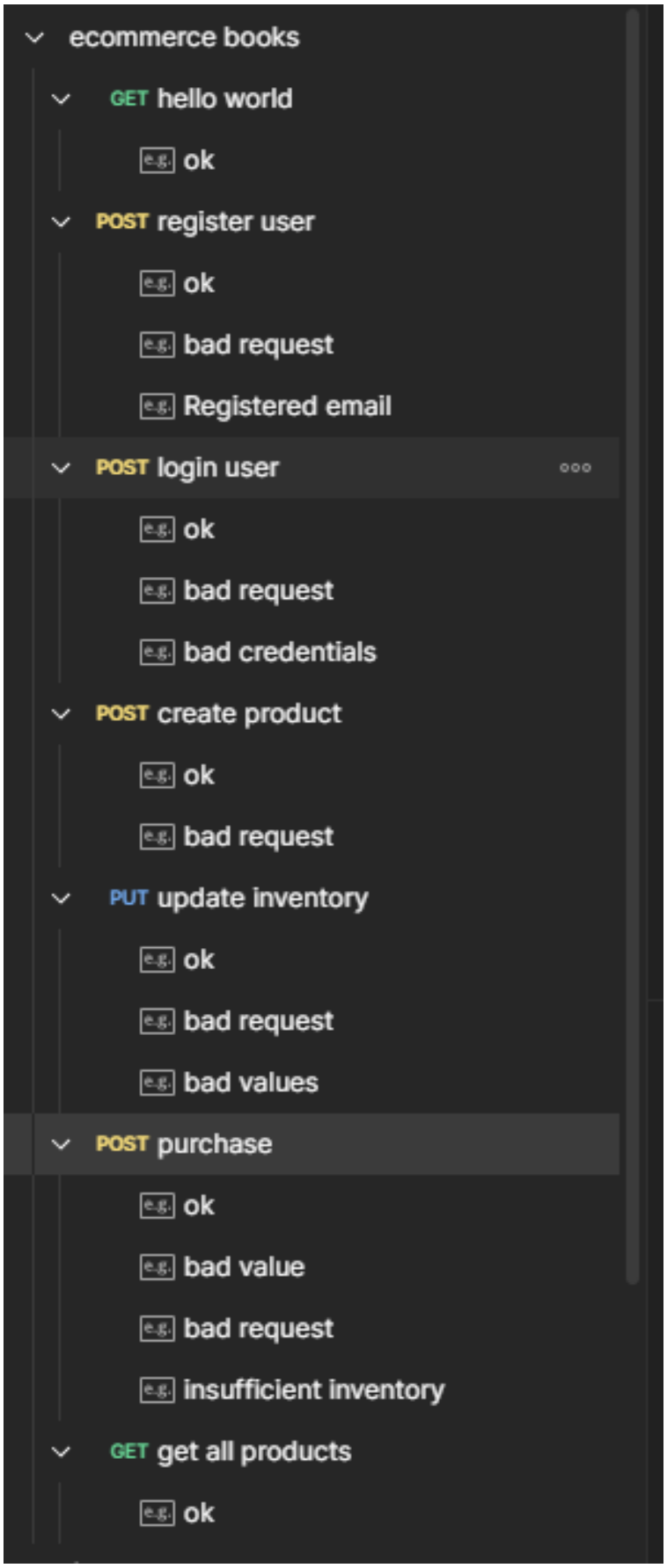

Figure 14: TI postman execution

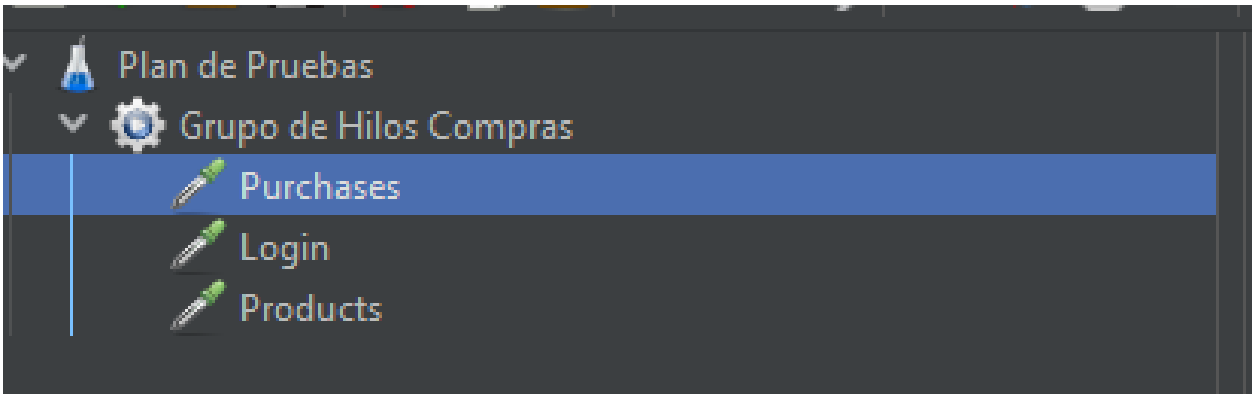

Figure 15: Stress Test