Checkpoint 1. Initial Setup & 2-Tier Architecture (Retail Management System): Documentation

Github Repository

https://github.com/alisavictory7/Retail-Management-System

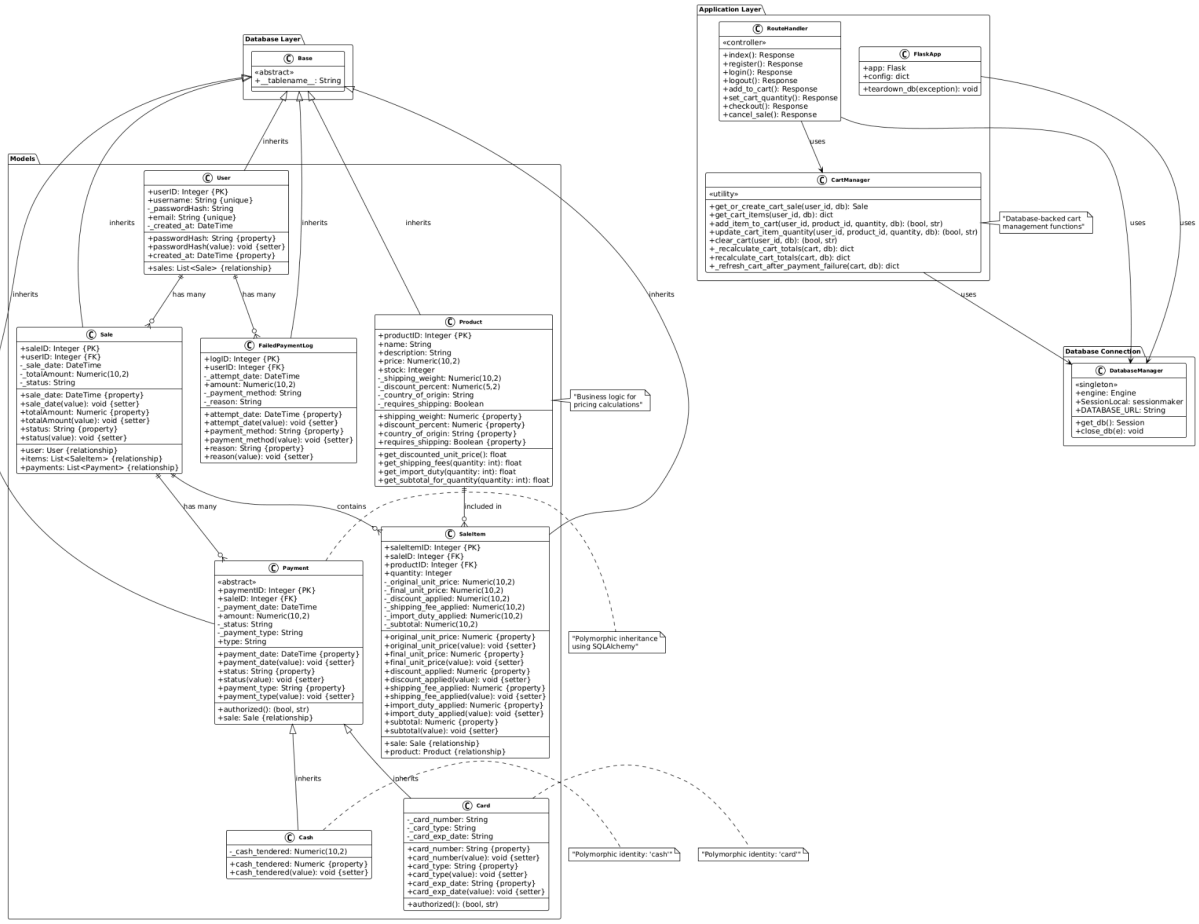
Link to Video

https://drive.google.com/file/d/1kOFm6N81PHJ_IYqKgdqIIJ_nOX2QIU1z/view?usp=drive_link

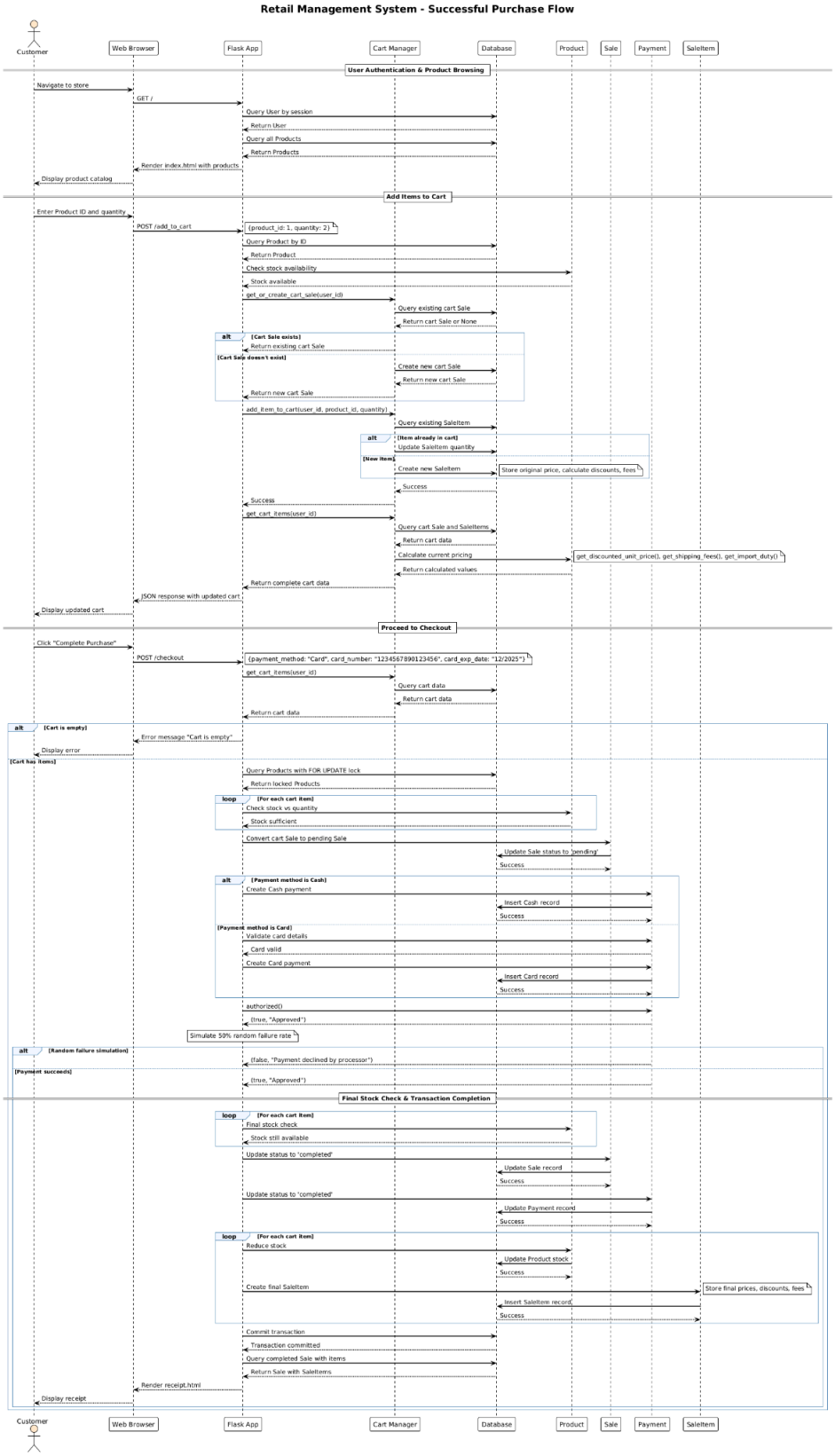
Part 3. UML

(1) Logical: Class diagram.

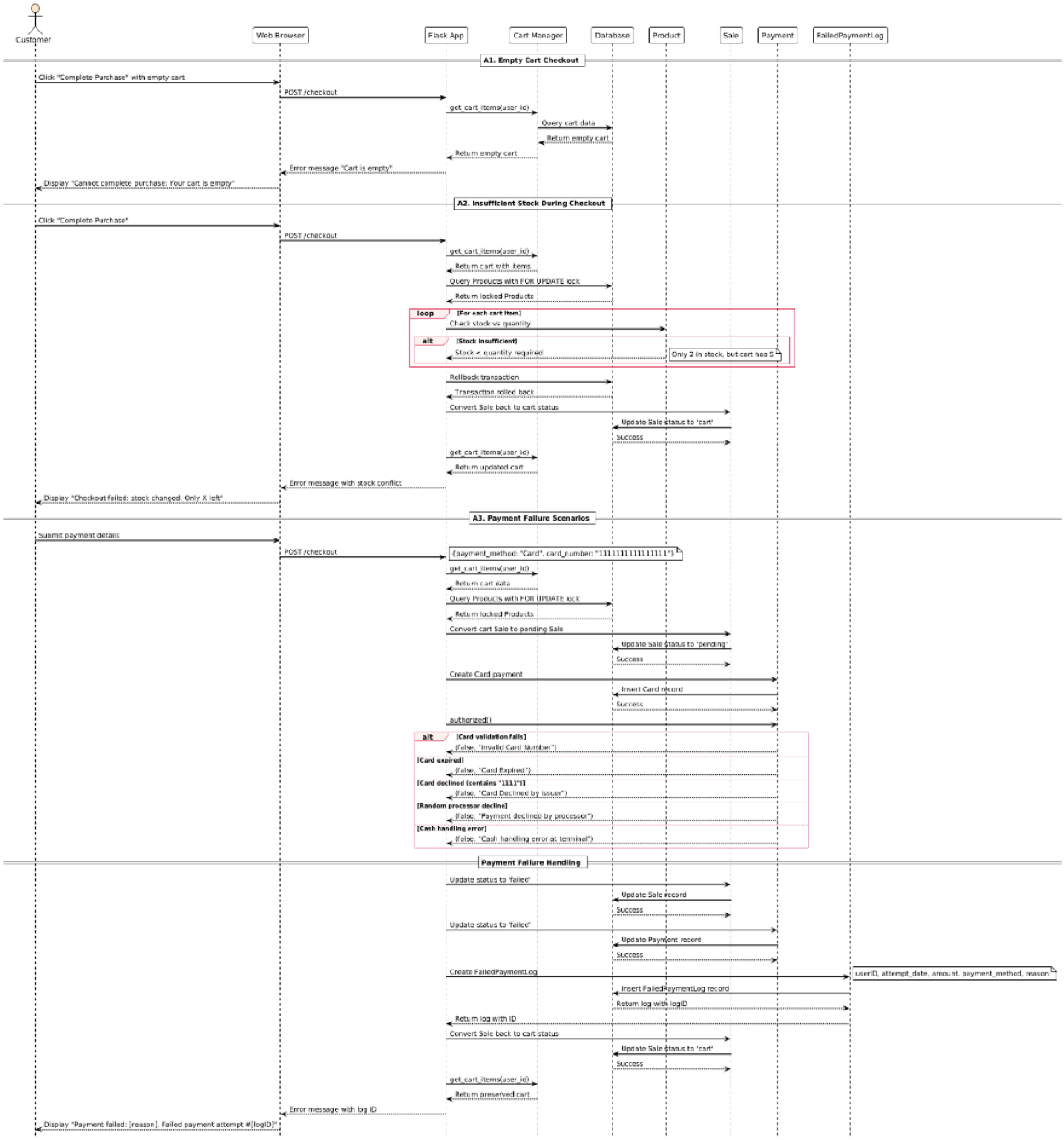
This diagram models the system's classes, their attributes, methods, and the relationships between them.

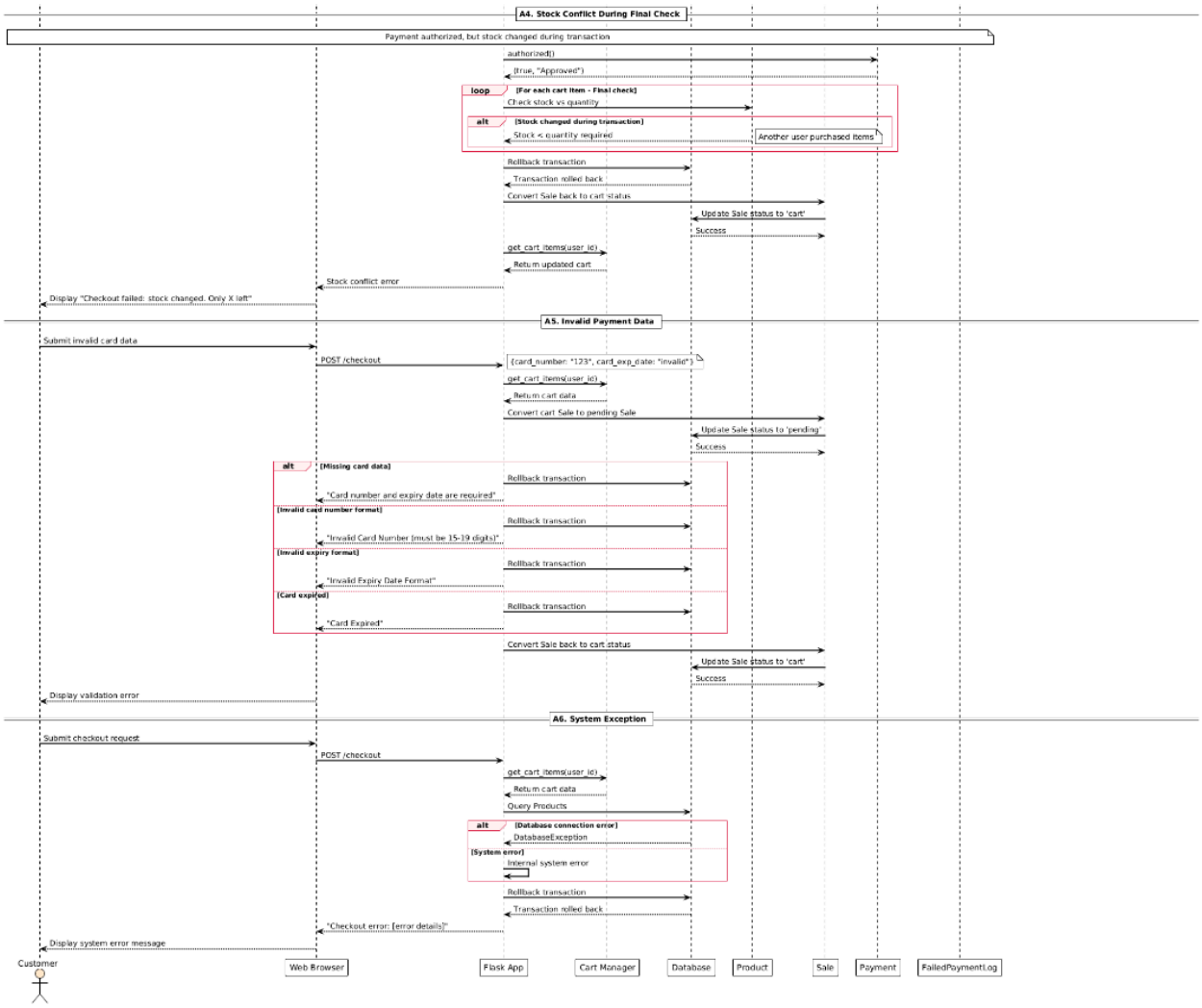


(2) Process: System Sequence diagram (purchase flow described above).

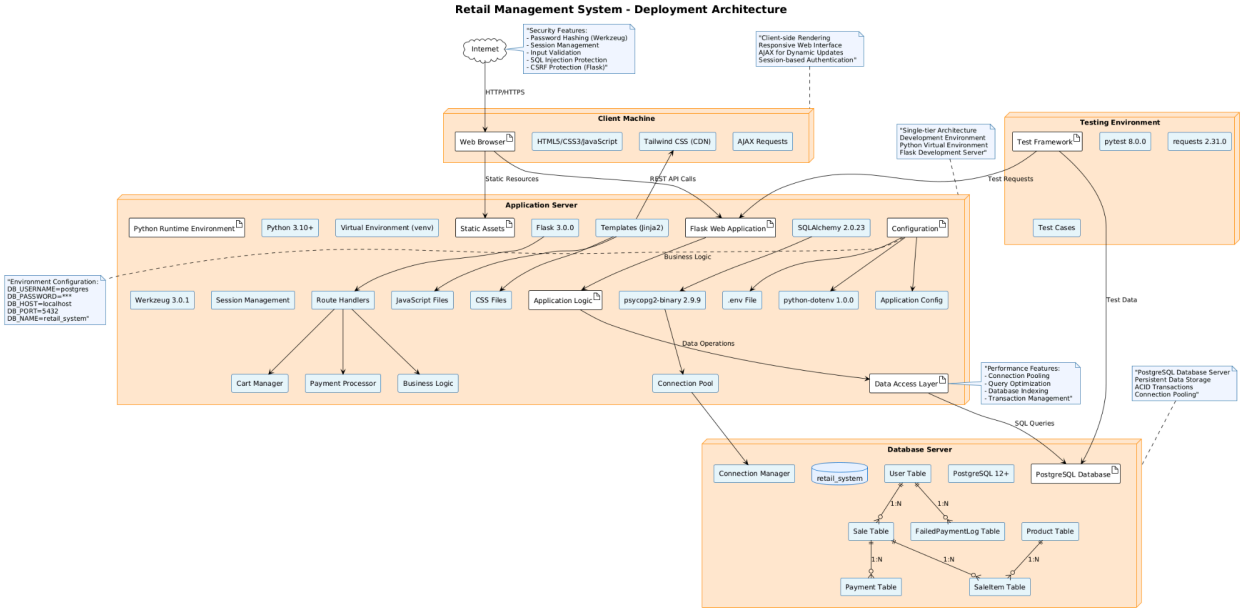


Retail Management System - Exception Scenarios

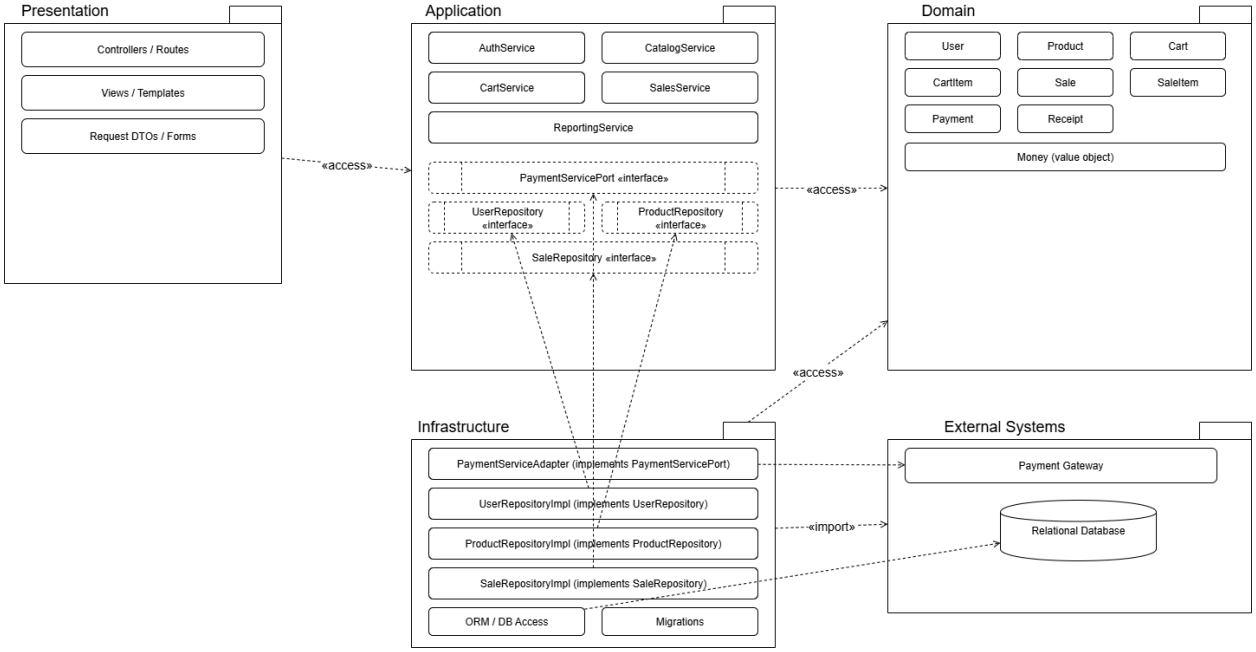



A4. Stock Conflict During Final Check
Payment authorized, but stock changed during transaction
authorized()
(true, "Approved")
loop
[For each cart item - Final check]
Check stock vs quantity
alt
[Stock changed during transaction]
Stock < quantity required
Another user purchased items
Rollback transaction
Transaction rolled back
Convert Sale back to cart status
Update Sale status to 'cart'
Success
get_cart_items(user_id)
Return updated cart
Stock conflict error
Display "Checkout failed: stock changed. Only X left"
A5. Invalid Payment Data
Submit invalid card data
POST /checkout
{card_number: "123", card_exp_date: "invalid"}
get_cart_items(user_id)
Return cart data
Convert cart Sale to pending Sale
Update Sale status to 'pending'
Success
alt
[Missing card data]
Rollback transaction
"Card number and expiry date are required"
[Invalid card number format]
Rollback transaction
"Invalid Card Number (must be 15-19 digits)"
[Invalid expiry format]
Rollback transaction
"Invalid Expiry Date Format"
[Card expired]
Rollback transaction
"Card Expired"
Convert Sale back to cart status
Update Sale status to 'cart'
Success
Display validation error
A6. System Exception
Submit checkout request
POST /checkout
get_cart_items(user_id)
Return cart data
Query Products
alt
[Database connection error]
DatabaseException
[System error]
Internal system error
Rollback transaction
Transaction rolled back
"Checkout error: [error details]"
Display system error message
Customer
Web Browser
Flask App
Cart Manager
Database
Product
Sale
Payment
FailedPaymentLog

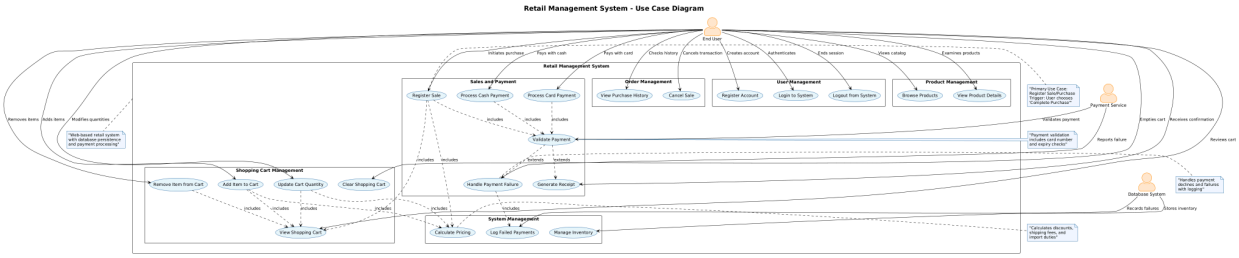
(3) Deployment: Client + DB.



(4) Implementation: Package/module diagram.



(5) Use-Case: Register Sale, and all other use cases you implemented.



Part 4. Architectural Decision Records (ADRs)

(1) Database Choice: SQLite vs. PostgreSQL

Status: Accepted

Context: The core requirement of this project is to implement a 2-tier retail prototype with a local persistence layer. A reliable database solution is needed that supports the "Register a Sale" use case, including transactionality and data persistence. The two primary options available were SQLite and PostgreSQL. While SQLite is simpler for local development, it lacks the features and robustness of a production-grade database. For this project, it was decided to prioritize a robust, scalable, and feature-rich foundation.

Decision: PostgreSQL will be used as the database. It provides a more accurate representation of a production environment.

Consequences:

Table 1. Pros and Cons of SQLite and PostgreSQL

	Pros	Cons
SQLite	• **Zero Configuration:** SQLite is a serverless, self-contained file-based database, meaning there's no separate server process to install or manage. This makes it incredibly easy to set up for a local, native project. • **Simplicity:** The entire database is a single file, which simplifies project setup and distribution. • **Built-in Support:** For Python, SQLite is built into the standard library.	• **Concurrency Limitations:** SQLite has limited support for concurrent writes. Only one write operation can happen at a time, which can be a bottleneck in multi-user environments. For this checkpoint, this isn't an issue, but it's a critical consideration for future scalability. • **Not a True Client-Server Model:** It doesn't use a client-server architecture, which is a key part of most enterprise applications. It's not a direct representation of a production environment. • **Limited Features:** It lacks some advanced features of client-server databases like user management, detailed permissions, and some complex SQL features.
PostgreSQL	• **Data Integrity and Concurrency:** PostgreSQL's robust transaction support (ACID compliance) is crucial for ensuring that `Sale` and `Stock` updates are atomic and reliable, especially in the event of a concurrency conflict. • **Scalability:** The client-server architecture of PostgreSQL provides a	• **Increased Setup Complexity:** PostgreSQL requires a separate server installation and configuration, which adds a layer of complexity to the initial project setup compared to SQLite. • **Higher Resource Usage:** A full-fledged database server consumes more system resources compared to a lightweight, file-based

	foundation that can scale from a local prototype to a multi-user application. ● **Feature Richness:** Access to advanced SQL features, roles, and permissions will be beneficial for future checkpoints. ● **Extensibility:** It offers a rich feature set, including advanced data types, indexing, and support for procedural languages, which can be beneficial for future project enhancements.	alternative. Hence there may be a trade-off with the performance quality attribute.

Alternatives Considered:

- **SQLite:** Rejected due to limited concurrency support and its lack of a client-server architecture, which does not align with the goal of building a robust, enterprise-like system.

(2) Persistence Style: ORM vs. DAO

Status: Accepted

Context: A persistence abstraction layer is required to handle interactions between the application logic and the chosen PostgreSQL database. The two main options were using a Data Access Object (DAO) pattern with a database driver or an Object-Relational Mapper (ORM). An ORM was chosen to streamline development and abstract away the low-level database operations.

Decision: An ORM will be used to manage database interactions. The specific ORM will be chosen based on the selected programming language (e.g., SQLAlchemy for Python, JPA/Hibernate for Java, Prisma for Node.js).

Consequences:

Table 2. Pros and Cons of ORM and DAO

	Pros	Cons
Object -Relational Mapping (ORM)	• **Enhanced Code Cohesion:** The ORM abstracts the complexity of SQL, which reduces the likelihood of syntax errors, typos, and other common mistakes that can occur with manual query writing. The type-safe nature of many ORMs also helps catch errors at compile-time. • **Maintainability:** By abstracting the database layer, the application code will be cleaner and more focused on business logic. Changes to the database schema can be managed more easily. • **Abstraction and Portability:** It provides a strong layer of abstraction between the application's business logic and the database. Thus this also improves code portability if the underlying database needs to be changed in the future. • **Reduced Boilerplate:** It automatically handles tasks like table creation and schema migration, saving development time.	• **Learning Curve:** There is an initial learning curve to understand the chosen ORM's conventions and best practices. • **Potential Performance Issues:** While generally efficient, complex or poorly written ORM queries can lead to performance bottlenecks. It is important to be mindful of how the ORM translates code to SQL. For very complex or highly optimized queries, an ORM might introduce overhead or make it difficult to write the most efficient SQL.
Data Access Object (DAO)	• **Explicit Control:** The DAO pattern gives you direct control over the SQL queries. This allows	• **More Boilerplate Code:** You will have to write more manual code for common database

	for fine-tuning performance and writing very specific, complex queries that might be difficult to express with an ORM. ● **No Framework Dependency:** You don't need to rely on a third-party framework; you use a simple database driver (e.g., `sqlite3` for Python, JDBC for Java) to execute your SQL. ● **Clear Separation:** The pattern enforces a clear separation between data access logic and business logic.	operations (e.g., CRUD—Create, Read, Update, Delete). This can be time-consuming and prone to errors. ● **Vendor Lock-in:** In this case, when writing raw SQL, the code might be less portable across different database systems.

Alternatives Considered:

- **Data Access Object (DAO):** Rejected because it would require writing and maintaining raw SQL queries for every database interaction, increasing development time and code complexity for this stage of the project.

Part 5. Testing (Local) Explanations

The following outlines the testing strategy for the Retail Management System and provides instructions on how to run the automated tests. The tests are written using the `pytest` framework, a standard for testing Python applications.

How to run

The tests are designed to be run from the command line. The following steps should be followed:

1. **Activate the Virtual Environment:** It is essential that the project's virtual environment is active. This ensures that `pytest` and all other project dependencies are available.

On Windows
.\venv\Scripts\activate

On macOS/Linux
source venv/bin/activate

2. **Navigate to the Project Root:** The test command should be run from the root directory of the project (the `Retail-Management-System` folder).
3. **Execute Pytest:** The `pytest` command is used to automatically discover and run all test files (files named `test_*.py`).
   pytest

Upon execution, `pytest` will run the tests and provide a summary of the results. A successful run will show that all tests have **PASSED**.

Explanation of test cases

The project includes two types of tests to ensure different aspects of the application are working correctly: a unit test for business logic and an integration test for system components.

2.1. Business Logic Unit Test (`tests/test_logic.py`)

Product Business Logic Tests:

- Pricing calculations: Discounted unit prices, subtotals for quantities
- Shipping fees: Based on weight and shipping requirements (including digital products)
- Import duties: Based on country of origin (USA = 0%, China = 5%)
- Multiple product types: Physical products, digital products, products with/without discounts

Payment Business Logic Tests:

- Cash payments: Always authorized
- Card payments: Validation for card numbers, expiry dates, declined cards
- Authorization logic: Tests for valid cards, invalid numbers, expired cards, declined cards

Cart Calculation Logic Tests:

- Grand total calculations: Including subtotals, shipping fees, and import duties
- Complex scenarios: Products with discounts, shipping, and import duties

Sale and User Business Logic Tests:

- Sale item calculations: Original prices, final prices, discounts, fees
- User properties: Username, email, password hash, creation date

2.2. Database Integration Test (`tests/test_integration.py`)

User Management Integration Tests:

- Registration and login flow: Complete user lifecycle
- Duplicate user handling: Prevents duplicate usernames
- Database verification: Ensures data persistence

Cart Management Integration Tests:

- Add to cart: With proper calculations and stock validation
- Insufficient stock handling: Returns appropriate errors
- Quantity updates: Cart recalculation after quantity changes

Payment Processing Integration Tests:
- Cash payment flow: Complete transaction with inventory updates
- Card payment validation: Server-side validation with proper error messages
- Payment failure handling: 50% random failure simulation

Database Transaction Integration Tests:
- Cart recalculation: Tests the _recalculate_cart_totals function with real database
- Stock conflict handling: Simulates concurrent access scenarios
- Transaction rollback: Ensures data consistency

Session Management Integration Tests:
- Cart persistence: Maintains cart across page navigation
- Logout functionality: Clears session data properly