E-commerce API Notes For Evaluation Criteria

Architectural Design and Technology Selection

To meet the defined requirements, a **Layered Architecture** model was chosen. This approach separates responsibilities into distinct layers, improving maintainability, scalability, and testability.

Chosen Technologies

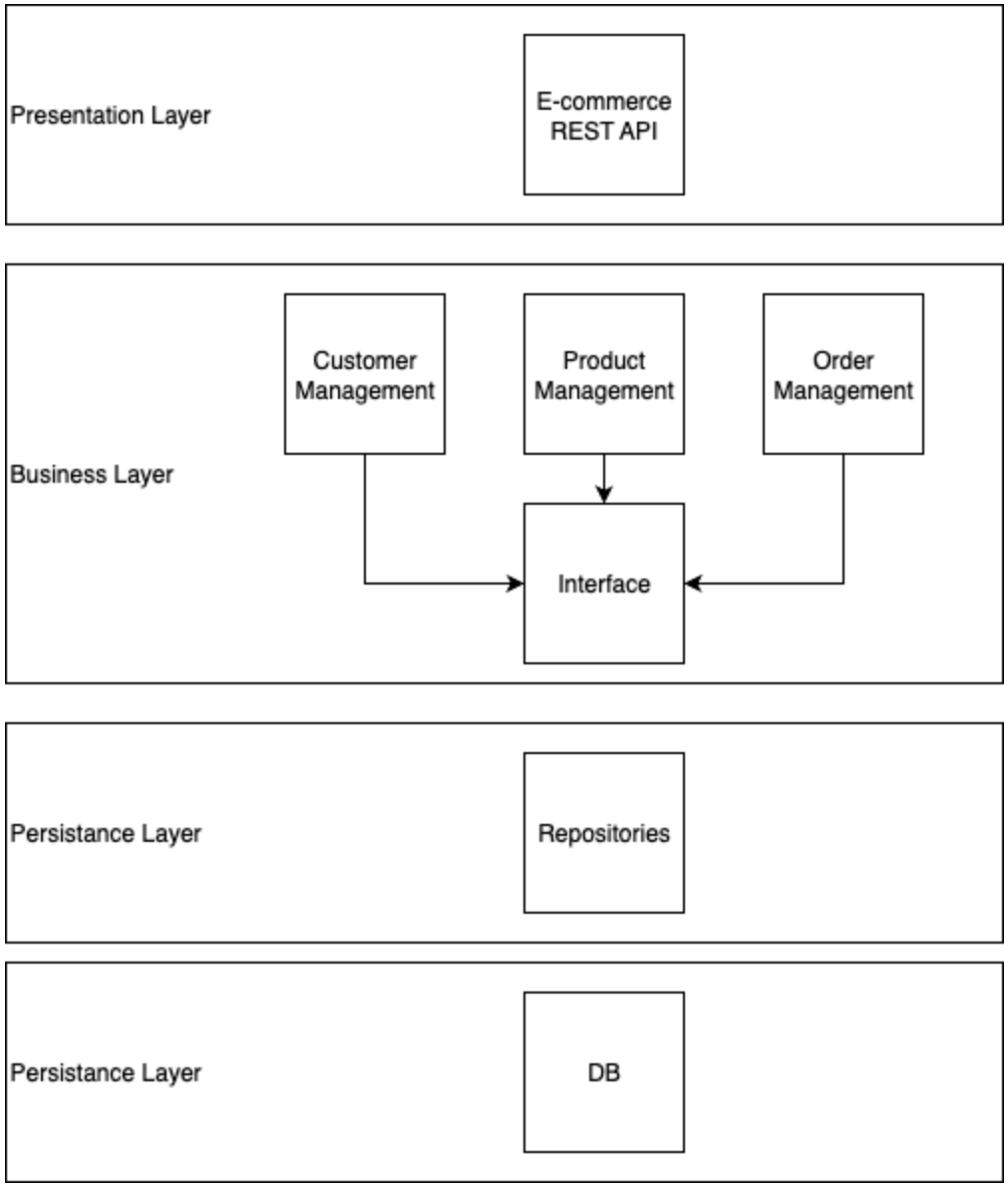
- **FastAPI (Python)** – Already selected by the project owner and fully supported by me, FastAPI was chosen for its excellent benchmark performance, ease of use in Python, rich library ecosystem, and advanced features such as automatic API documentation.
- **MySQL – Also selected by the project owner and fully supported by me, as it ensures reliable handling of non-idempotent operations, maintains strict structural data integrity, and provides full ACID compliance.**
- **Redis** – Used both as a **queue system** (for background processing via Celery) and as a **cache layer** to improve response time and reduce database load.
- **k6** – Used for **performance testing** to ensure system stability under high load.

Problem Analysis and Design Patterns

The system is structured around four primary layers:

1. **Presentation Layer** – Handles HTTP requests through FastAPI.
2. **Business Layer** – Contains core domain logic (Customer, Product, and Order Management).
3. **Persistence Layer** – Includes repository classes that manage data access.
4. **Database Layer** – Manages the physical data storage (MySQL).

In the **Business Layer**, three core components (Customer, Product, and Order Management) communicate through a central **Interface**. This design ensures loose coupling and makes the system easier to extend or modify. If one component grows or changes, others remain unaffected.



Applied Design Patterns

- **Layered Architecture** - Provides a clear separation of concerns and supports scalability.

- **Interface** - Allows shared access to business logic between components without creating dependencies. This enables future adaptation to microservices through an API Gateway.
- **Builder Pattern** - Used for builder classes for db objects to standardize object modifications and ensure that required fields are standardize correctly.
- **Asynchronous Task Queue** - Implemented with Redis and Celery to handle background jobs safely and efficiently, ensuring that no data is lost even if the application restarts.

Measurable Achievements

REQ-3.1.1 – Performance Test

To validate the performance requirement, **k6** was used to simulate 1000 requests per second (RPS) for 10 seconds against the GET /products/{product_id} endpoint.

Results showed that **99% of all requests completed under 100 ms (p(99)=45.46 ms)**, fully meeting the REQ-3.1.1 target. **No failed requests or timeouts occurred during the test.**

REQ-3.2.1 – Race Condition Test

Using a Python concurrency test script (req-3-2-1_test_race_condition.py), 100 simultaneous order creation requests were sent for a product with stock quantity = 1.

The system successfully handled the race condition: **exactly 1 order completed** (COMPLETED), while **99 orders failed** due to insufficient stock, and the final stock in the database correctly reduced to **0**.

REQ-3.2.2 – High Volume Test

The high-volume scenario was tested with **10,000 concurrent order requests** using **k6**.

The API successfully accepted **6,472 requests** without any 5xx or network errors, maintaining consistent order creation and balance updates.

Post-test verification confirmed:

- Customer wallet balance reduced from 10,001 to **3,529**
- Orders table contained **6,472 records**
- Product stock reduced from 10,000 to **3,528**

> A high-volume scenario was tested using k6 with a target load of 1000 requests per second (RPS) for 10 seconds, totaling 10,000 requests, of which 6,472 were successfully processed while 3,528 were dropped or delayed due to system limits. This resulted in an effective throughput of approximately 647 RPS (65% of the target rate), with no 5xx or network errors observed, demonstrating stable server behavior under stress.

REQ-3.3.1 – API Response Time Validation

Both **CreateOrderRequest** and **CancelOrderRequest** endpoints were tested to verify that they respond within **200 milliseconds**, without waiting for background processes (such as queue handling or asynchronous workers) to complete.

Results:

- **CreateOrderRequest:** 0.059252 s (**≈ 59 ms**)
- **CancelOrderRequest:** 0.075104 s (**≈ 75 ms**)

Both endpoints successfully returned their respective order_id values within the required time frame, confirming that **REQ-3.3.1 has passed**.