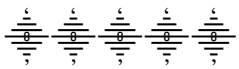
ĐẠI HỌC PHENIKAA

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN PHENIKAA



BÁO CÁO BÀI TẬP LỚN

HỌC PHẦN KIẾN TRÚC PHẦN MỀM

Đề tài: Shophere-e-commerce platform

Thành viên: Phạm Tuấn Anh-23010724

Giáo viên hướng dẫn: Vũ Quang Dũng

Lớp tín chỉ: Kiến trúc phần mềm-1-2-25(N02)

1. Lab Specific Section: I. Requirements Elicitation & Modeling

This section documents the deliverables for Lab 1, focusing on the software requirements and the visual modeling of the system context.

1.1 Software Requirements Specifications (SRS)

Based on the ShopSphere scenario, the software requirements are defined as follows:

1.1.1 System Actors

The system interacts with the following primary entities:

- **Web Customer:** End-users who browse products, manage carts, and make purchases.
- **Administrator:** Internal users who manage product catalogs and process orders.
- **Payment Gateway:** An external system responsible for processing financial transactions securely.

1.1.2 Functional Requirements

- **For Web Customer:**
    - Register and Login.
    - Search and Browse Product Catalog.
    - Manage Shopping Cart (Add/Remove items).
    - Make Purchase (Checkout).
    - Manage Personal Profile.
- **For Administrator:**
    - Product Management (Create, Read, Update, Delete products).
    - Order Processing (View and update order status).

1.1.3 Architecturally Significant Requirements (ASRs)

The following ASRs have been identified as critical drivers that will influence the system architecture:

ASR ID	Quality Attribute	Description	Architectural Impact
ASR -1	**Security**	The system must protect sensitive user data (PII) and financial information. Only authenticated users can access personal profiles or admin functions.	**Authentication & Authorization:** Requires dedicated security modules (e.g., JWT, HTTPS) at service entry points.
ASR -2	**Performance**	The system must support a high volume of concurrent users browsing the catalog without performance degradation.	**Separation of Concerns:** The Product Service should be decoupled to allow independent scaling or caching strategies.
ASR -3	**Availability**	The user's shopping cart state must be persisted across sessions. Data should not be lost if the browser is closed.	**State Management:** Requires a robust persistence mechanism for session data, separating UI logic from data storage.

1.2 Modeling Artifact: UML Use Case Diagram

The following diagram illustrates the system boundaries, actors, and major use cases.

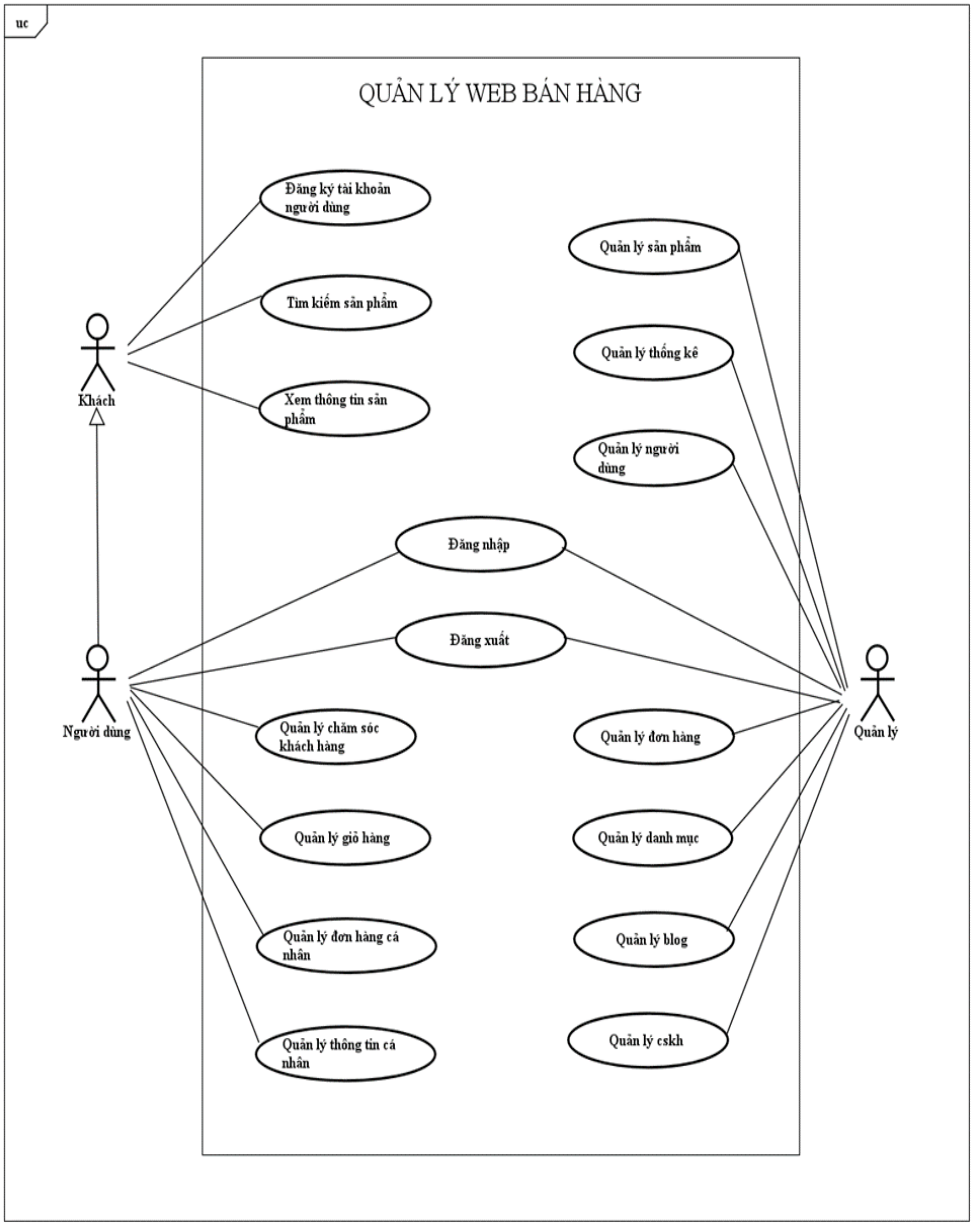


Diagram Description: The UML Use Case Diagram visualizes the functional scope of ShopSphere. It depicts the **Web Customer** interacting with core features like "Search Catalog" and "Make Purchase," while the **Administrator** handles "Product Management."

A critical part of the model is the **"Make Purchase"** use case, which includes the **"Secure Payment"** use case (mandatory interaction with the Payment

Gateway) and is extended by the **"Apply Coupon"** use case (optional behavior). This structure clearly defines the system boundaries and external dependencies.

2. Architectural Analysis of ASRs

This section analyzes how the identified ASRs will map to the Layered Architecture in the upcoming design phase.

- **ASR-1 (Security) → Business Logic Layer:** Security policies and validation logic must be enforced within the Business Logic Layer. This ensures that unauthorized access is blocked regardless of whether the request comes from the Web UI or another source, protecting the core services effectively.
- **ASR-2 (Performance) → Persistence & Business Layers:** To meet performance goals, the architecture must allow the Business Logic layer to implement caching or efficient data retrieval strategies from the Persistence layer without affecting the Presentation layer. This separation allows the catalog read operations to be optimized independently.

3. Activity Practice 1: Defining Layers and Responsibilities

3.1 Definition of Four Layers

The ShopSphere system adopts a strict Layered Architecture organized into four logical layers to ensure separation of concerns:

Layer	Purpose / Responsibility	Output / Artifact
1. Presentation Layer (UI/API)	Handles incoming HTTP requests from the Web Customer, performs input validation, and formats the JSON response. It does not contain business logic.	**Artifact:** Flask Routes / Controllers (e.g., `product_routes.py`)

2. Business Logic Layer	Contains the core business rules of ShopSphere (e.g., checking stock availability, calculating discounts). It orchestrates data flow between the UI and Persistence layers.	**Artifact:** Service Classes (e.g., ProductService)
3. Persistence Layer (Data Access)	Abstracts the database interactions. It performs CRUD operations (Create, Read, Update, Delete) without exposing SQL details to upper layers.	**Artifact:** Repository Classes (e.g., ProductRepository)
4. Database Layer	The physical storage system responsible for persisting data.	**Artifact:** SQLite Database (Schema: Products table)

4. Activity Practice 2: Identifying Components (Product Catalog)

Based on the "Search & Browse Catalog" functionality defined in Lab 1, the following components are identified for implementation:

4.1 Component Identification

- **Presentation Component:** ProductController
  - *Role:* Receives GET /products requests, parses query parameters (like category or price range), calls the Service layer, and returns the product list as JSON.
- **Business Component:** ProductService
  - *Role:* Implements business logic (e.g., get_all_products()), ensures only active products are shown, and applies business rules.
- **Persistence Component:** ProductRepository

- *Role:* Executes database queries (e.g., SELECT * FROM products) and maps the raw database rows to Product objects.

5. Activity Practice 3: Component Diagram Modeling

5.1 Logical View Diagram

The following diagram illustrates the system's static structure and component dependencies.

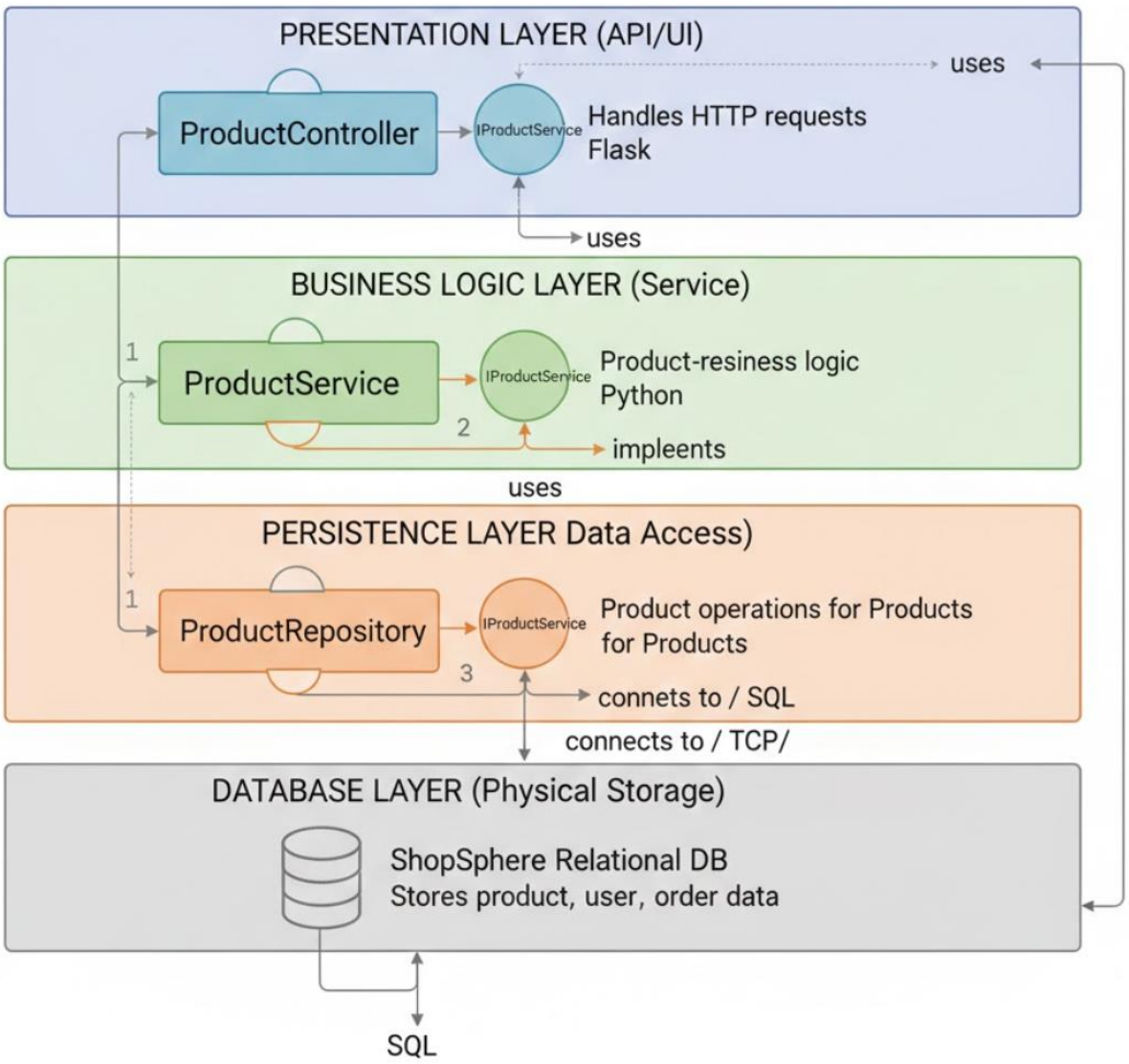


Diagram Description:

The C4 Component Diagram illustrates the internal structure of the **ShopSphere Backend API** using a strict **4-Layer Architecture**:

1. **Presentation Layer (API/UI):**

    - **Component:** ProductController.
    - **Interaction:** It handles incoming HTTP requests from the Web/Mobile App.
    - **Dependency:** It has a "uses" relationship pointing downwards to the IProductService interface in the layer below.
2. **Business Logic Layer (Service):**
    - **Component:** ProductService.
    - **Interaction:** This component encapsulates the core domain logic (e.g., checking active status, calculating prices).
    - **Interfaces:**
        - It **provides** (implements) the IProductService interface (shown as a lollipop symbol) for the Controller to use.
        - It **requires** (uses) the IProductRepository interface to access data.
3. **Persistence Layer (Data Access):**
    - **Component:** ProductRepository.
    - **Interaction:** This component abstracts the SQL logic.
    - **Interfaces:**
        - It **provides** (implements) the IProductRepository interface for the Service layer.
        - It connects directly to the underlying physical database using a database driver (e.g., via TCP/IP).
4. **Database Layer (Physical Storage):**
    - **Component:** ShopSphere Relational DB.
    - **Interaction:** Stores physical data tables (Products, Users, Orders). It receives SQL queries from the Repository layer and returns raw data rows.

Architectural Enforcement:

The diagram demonstrates **Strict Layering**: Dependencies flow only downwards (Presentation $\rightarrow$ Business $\rightarrow$ Persistence $\rightarrow$ Database). This ensures that the Controller is never aware of the Database, achieving high **Separation of Concerns** and **Modifiability**.

6. Architectural Justification (Mapping from Lab 1 ASRs)

The Layered Architecture is selected to satisfy the Architecturally Significant Requirements (ASRs) identified in Lab 1:

ASR ID	ASR Description (Lab 1)	How Layered Architecture Satisfies It
ASR -1	**Security** (Protect sensitive data & access control)	Security policies are centralized in the **Business Logic Layer**. This ensures that every request, whether from a web browser or a mobile app, must pass through the same security checks in the Service layer before accessing data.
ASR -2	**Performance** (High volume browsing)	The separation of the **Persistence Layer** allows us to optimize database queries or implement caching (e.g., Redis) within the Repository without changing the Presentation Layer code, maintaining high performance for the Product Catalog.
ASR -3	**Availability** (Cart state persistence)	The strict separation allows the **Database Layer** to be managed independently. We can implement database replication or failover strategies to ensure data availability without rewriting the application logic in the upper layers.

7. System Architecture & Technologies

To ensure modularity and maintainability, the system is built using the following stack:

- **Language:** Python 3
- **Framework:** Flask (Web Framework)
- **Database:** MySQL (XAMPP)
- **Pattern:** 4-Layer Architecture (Presentation, Business, Persistence, Database)

Architectural Layers:

1. **Presentation Layer (Controller):** Handles HTTP requests and renders HTML templates.
2. **Business Logic Layer (Service):** Contains domain logic (e.g., filtering available products).
3. **Persistence Layer (Repository):** Handles direct SQL database operations.
4. **Database Layer:** Physical storage (MySQL webbandocu database).

8. Project Structure

The source code is organized into distinct packages to reflect the architecture physically:

```
ShopSphere/
│
├── app.py                # Entry Point & Dependency Injection wiring
├── database.py           # MySQL Connection Configuration
├── templates/
│   └── index.html        # UI Representation (HTML/Jinja2)
├── static/
│   ├── css/              # Stylesheets
│   └── assets/           # Product Images
│
├── controllers/          # PRESENTATION LAYER
│   └── product_controller.py
├── services/             # BUSINESS LOGIC LAYER
```

```
│   └── product_service.py
├── repositories/           # PERSISTENCE LAYER
│   └── product_repository.py
└── models/                 # SHARED ENTITIES
    └── product.py
```

9. Implementation Details

9.1. Domain Model (models/product.py)

This class maps directly to the products table in the database.

```
class Product:
    def __init__(self, item_id, item_name, item_price, item_image,
item_quantity):
        self.item_id = item_id
        self.item_name = item_name
        self.item_price = item_price
        self.item_image = item_image
        self.item_quantity = item_quantity

    # Helper to format data for the View
    def to_dict(self):
        return {
            "id": self.item_id,
            "name": self.item_name,
            "price": f"{self.item_price:,.0f} đ",
            "image": self.item_image,
            "stock": self.item_quantity
        }
```

9.2. Persistence Layer (repositories/product_repository.py)

This layer executes raw SQL queries. It is the only layer that knows about the Database Driver.

```
from database import get_db_connection
from models.product import Product

class ProductRepository:
```

```
    def find_all(self):
        conn = get_db_connection()
        if not conn: return []

        cursor = conn.cursor(dictionary=True)
        query = "SELECT item_id, item_name, item_price, item_image,
item_quantity FROM products"
        cursor.execute(query)
        rows = cursor.fetchall()

        # Mapping SQL rows to Product Objects
        products = []
        for row in rows:
            products.append(Product(
                item_id=row['item_id'],
                item_name=row['item_name'],
                item_price=row['item_price'],
                item_image=row['item_image'],
                item_quantity=row['item_quantity']
            ))
        return products
```

9.3. Business Logic Layer (services/product_service.py)

This layer handles business rules. For this lab, the rule is to filter out out-of-stock items.

```
class ProductService:
    def __init__(self, product_repository):
        # Dependency Injection
        self.product_repository = product_repository

    def get_home_products(self):
        all_products = self.product_repository.find_all()
        # Logic: Only show products with quantity > 0
        active_products = [p for p in all_products if p.item_quantity > 0]
        return active_products
```

9.4. Presentation Layer (controllers/product_controller.py)

The controller coordinates the response without knowing the underlying logic details.

```
from flask import render_template

class ProductController:
    def __init__(self, product_service):
        self.product_service = product_service

    def index(self):
        products_list = self.product_service.get_home_products()
        return render_template('index.html', products=products_list)
```

10. Dependency Injection (app.py)

To adhere to the Inversion of Control principle, dependencies are injected at the application startup:

```
# Wiring the layers together
product_repo = ProductRepository()          # Create Repository
product_service = ProductService(product_repo) # Inject Repo into Service
product_controller = ProductController(product_service) # Inject Service into Controller
```

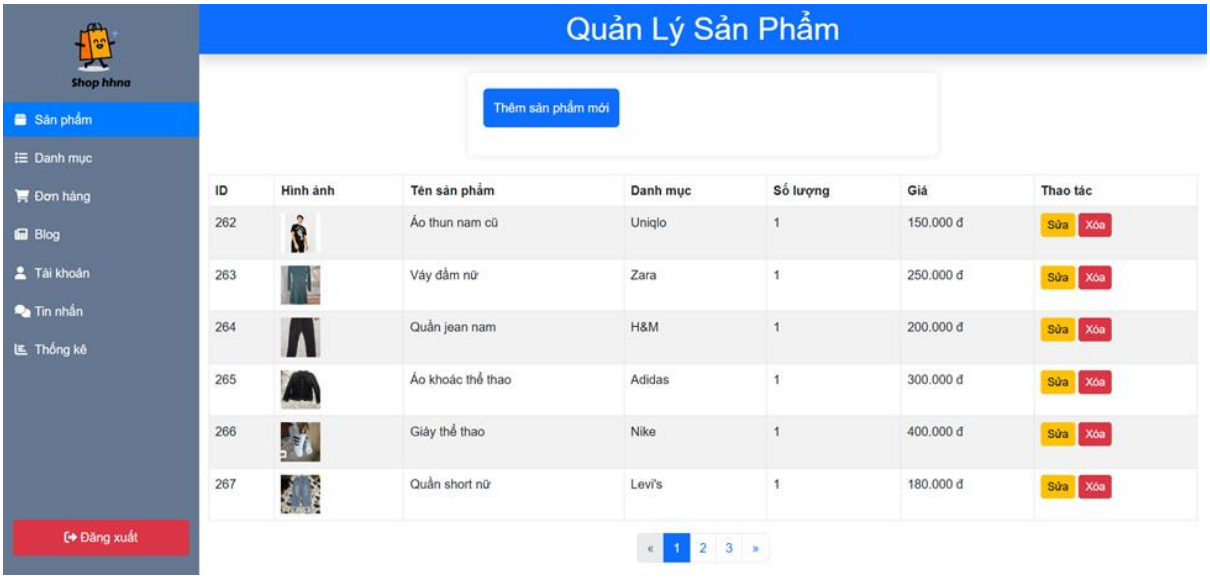
11. Execution & Results

11.1. Database Setup

The webbandocu database was imported successfully into MySQL. The products table contains the inventory data.

11.2. Application Interface

Upon running the Flask application, the system successfully retrieves data from MySQL, filters it via the Service layer, and displays it on the web interface.



12. Decomposition by Business Capability

Based on the core functionalities of an e-commerce platform for second-hand goods, we have identified and decomposed the system into the following independent microservices. Each service owns its data and logic.

Business Capability	Microservice Name	Responsibility	Owned Data (Entities)
Identity & Access	**Identity Service**	Manages user registration, authentication (Login), and profile management.	Users, Roles, Tokens
Catalog Management	**Catalog Service**	Manages product listings, categories, and inventory status (New/Old).	Products, Categories, Images

Buying & Checkout	**Order Service**	Handles shopping cart, order placement, and order history.	Orders, OrderDetails, Cart
Notifications	**Notification Service**	Sends emails/SMS for order confirmation and status updates.	Templates, Logs

13. Service Contracts

To ensure loose coupling, services interact only via standardized RESTful APIs. Below are the API definitions for the key services.

13.1. Catalog Service API

This service exposes endpoints for browsing products (The functionality implemented in Lab 3).

Method	Endpoint	Description	Request Body	Response
GET	/api/v1/products	Get list of all available products	N/A	200 OK (List JSON)
GET	/api/v1/products/{id}	Get details of a specific product	N/A	200 OK (Product JSON)

POST	/api/v1/products	Add a new second-hand item	{name, price, status...}	201 Created

13.2. Order Service API

This service handles the purchasing process.

Method	Endpoint	Description	Request Body	Response
POST	/api/v1/orders	Place a new order	{user_id, items:[]}	201 Created
GET	/api/v1/orders/{id}	View order status	N/A	200 OK

14. System Communication Design

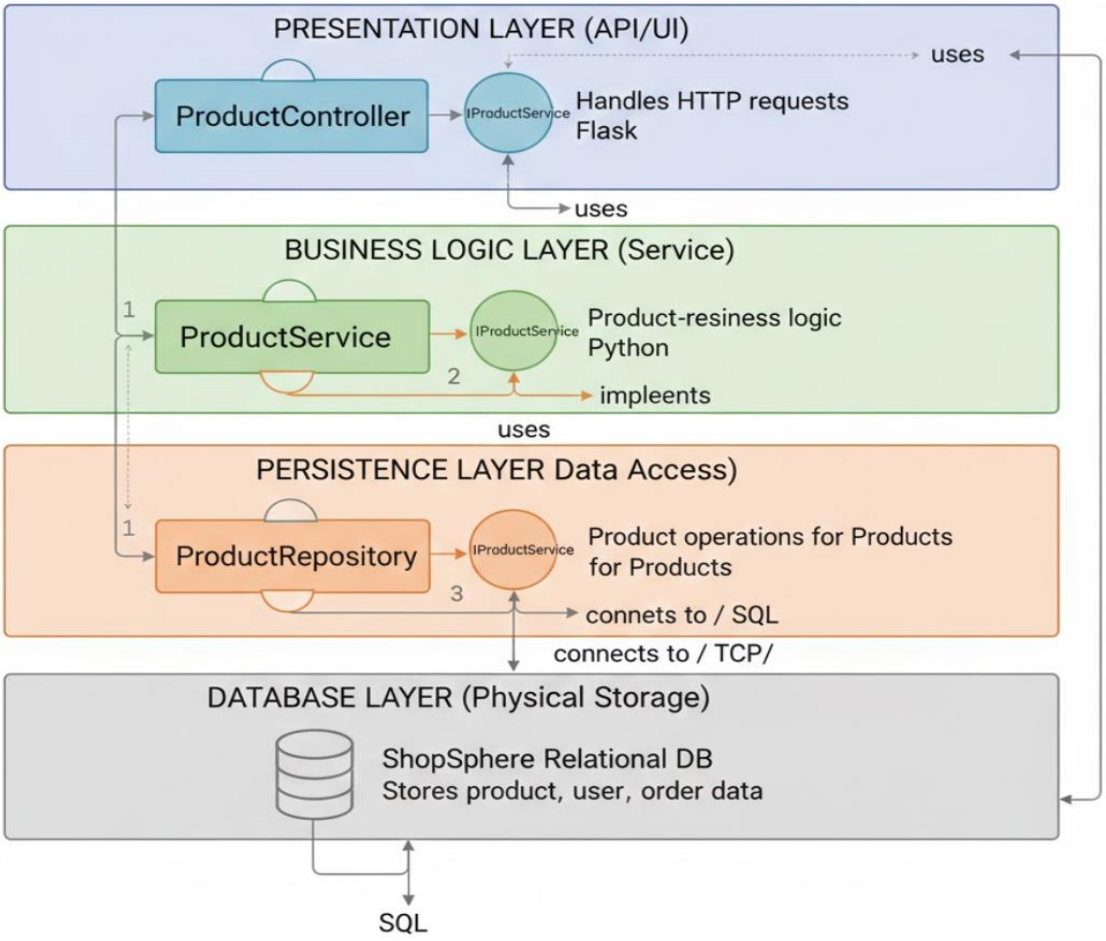
In the Microservices architecture, ShopSphere uses a hybrid communication strategy:

1. **Synchronous Communication (HTTP/REST):** Used for client-facing interactions where immediate feedback is required.
    - *Example:* When a user views the product list, the Frontend calls the **Catalog Service** via HTTP GET. The user waits until the data is returned.
2. **Asynchronous Communication (Message Queue):** Used for backend processing to decouple services and improve performance.
    - *Example:* When an order is placed successfully in **Order Service**, it publishes an event OrderPlaced to a Message Broker (e.g., RabbitMQ). The **Notification Service** consumes this event to send

an email. This ensures the user doesn't have to wait for the email to be sent before seeing the "Order Success" screen.

15. C4 Model - System Context

The following diagram illustrates the high-level interaction between the ShopSphere system and external actors.



16. Implementation

16.1. Project Structure

The microservice is organized into a separate directory named Product_Microservice, distinct from the main website's source code. This enforces physical separation of concerns.

16.2. Microservice Logic

Unlike the Layered Architecture in Lab 3 where logic was split across Controllers and Repositories, this Microservice encapsulates the necessary logic in a single entry point api.php to handle HTTP requests efficiently.

Key responsibilities of this file:

1. **Database Connection:** Connects securely to the webbandocu database.
2. **CORS Handling:** Allows external applications to request data.
3. **Routing:** Detects request methods (GET, POST) to trigger appropriate actions.

Code Snippet (Core Logic):

```
// Handling GET Request for Product List or Detail

if ($method == 'GET') {

    if (isset($_GET['id'])) {

        // Fetch specific product by ID

        $id = intval($_GET['id']);

        $sql = "SELECT * FROM products WHERE item_id = $id";

        // ... execute query and return JSON

    } else {

        // Fetch all products

        $sql = "SELECT item_id, item_name, item_price, item_quantity
FROM products";
```

```
            // ... execute query and return JSON array
        }
    }
```

17. Test Results (Isolation Testing)

To verify that the service operates independently, we bypassed the main web interface and interacted directly with the Microservice via **Port 5001**.

Startup Command:

```
php -S localhost:5001 api.php
```

17.1. Test: Get All Products (GET)

- **Endpoint:** http://localhost:5001/api.php
- **Description:** Retrieves the full catalog of second-hand items.
- **Response Status:** 200 OK

17.2. Test: Get Product Details (GET)

- **Endpoint:** http://localhost:5001/api.php?id=1
- **Description:** Retrieves details for a specific product (e.g., ID 1).
- **Response Status:** 200 OK

17.3. Test: Create New Product (POST)

- **Endpoint:** http://localhost:5001/api.php
- **Description:** Adds a new item to the inventory.
- **Payload (JSON):**

```
{
  "name": "Vintage Camera",
  "price": 1500000,
  "quantity": 1
}
```

- **Response Status:** 201 Created

18. Implementation

18.1. Architecture Overview

The Gateway runs independently on **Port 5000**. It intercepts all incoming traffic.

- Client sends request to localhost:5000 (Gateway).
- Gateway verifies the Security Token.
- If valid, Gateway uses cURL to call localhost:5001 (Product Service).
- Gateway returns the response to the Client.

18.2. Gateway Logic (**gateway.php**)

The gateway logic is divided into three main layers:

1. **Security Layer:** Checks for the presence of a valid Authorization header. We simulated two roles:
    - Bearer valid-user-token: Allows read-only access (GET).
    - Bearer admin-token: Allows write access (POST). If the token is missing or invalid, the Gateway returns HTTP 401.
2. **Routing Layer:** Identifies the destination service. In this implementation, it constructs the target URL pointing to localhost:5001 while preserving query parameters (e.g., ?id=1).
3. **Proxy Layer:** Uses curl_exec to forward the client's request (Method, Headers, Body) to the backend and returns the backend's response to the client. It also handles HTTP 503 errors if the backend service is offline.

19. Test Results

We verified the Gateway's functionality using **Postman** through four specific scenarios:

19.1. Security Test: Unauthorized Access

- **Scenario:** Client sends a request without an Authentication Token.
- **Request:** GET http://localhost:5000
- **Result:** The Gateway blocked the request and returned **HTTP 401 Unauthorized** with the message: *"Unauthorized access. Invalid or missing token."*

19.2. Routing Test: Authorized Access (Success)

- **Scenario:** Client sends a valid user token.
- **Request:** GET http://localhost:5000 with Header Authorization: Bearer valid-user-token.
- **Result:** The Gateway successfully forwarded the request to Port 5001. The response was **HTTP 200 OK**, containing the JSON list of products retrieved from the backend database.

19.3. Authorization Test: Forbidden Action

- **Scenario:** A regular User tries to perform an Admin action (Creating a product).
- **Request:** POST http://localhost:5000 with Header Authorization: Bearer valid-user-token.
- **Result:** The Gateway inspected the method and token, determining the user lacked permissions. It returned **HTTP 403 Forbidden** with the message: *"Forbidden. Only Admins can create products."*

19.4. Resilience Test: Service Unavailable

- **Scenario:** The Backend Service (Port 5001) was manually stopped to simulate a crash.
- **Request:** GET http://localhost:5000 with a valid token.
- **Result:** The cURL connection failed. Instead of crashing, the Gateway handled the exception and returned **HTTP 503 Service Unavailable** with the message: *"Service Unavailable. Backend is down."*

20. System Architecture Description

The communication flow implemented in this lab decouples the order placement process from the notification process:

1. **User Action:** A customer places an order via the Order Service.
2. **Producer (Order Service):** Instead of sending an email directly, the service serializes the order details into a JSON object and publishes it to a RabbitMQ queue named order_queue. The service then immediately responds to the user.
3. **Message Queue:** RabbitMQ acts as a buffer, storing the message persistently until a consumer is available.

4. **Consumer (Notification Service):** A separate background process continuously listens to order_queue. Upon receiving a message, it deserializes the data and executes the email sending logic (simulated with a time delay).

21. Implementation Details

21.1. Order Service (Producer)

The Producer script (producer.php) is responsible for connecting to the RabbitMQ server and dispatching events. Key implementation details include:

- Establishing a connection to localhost on port 5672.
- Declaring a durable queue named order_queue to ensure message persistence.
- Encoding order data (Order ID, Customer Email, Price) into JSON format.
- Publishing the message using basic_publish and immediately closing the connection to free up resources.

21.2. Notification Service (Consumer)

The Consumer script (consumer.php) runs as a persistent background worker. Its logic includes:

- maintaining a live connection to RabbitMQ.
- Defining a **callback function** to handle incoming messages.
- Within the callback, a sleep(2) function is used to simulate the latency of communicating with an external email server (e.g., SMTP).
- Acknowledging the message processing to remove it from the queue.

22. Test Scenarios and Results

To validate the architecture, we conducted two primary test scenarios using the Command Line Interface (CLI).

22.1. Scenario A: Asynchronous Performance Test

Objective: Prove that the User (Producer) does not wait for the Email (Consumer) to finish.

Execution:

1. We started the Consumer script in Terminal 1.
2. We ran the Producer script in Terminal 2.

Result:

- **Terminal 2 (Producer):** The script finished execution instantly (less than 0.1 seconds), confirming that the order placement is non-blocking.
- **Terminal 1 (Consumer):** Simultaneously, the consumer detected the message. It displayed "Received...", paused for 2 seconds (simulating work), and then displayed "Email sent".
- **Conclusion:** The system successfully demonstrated asynchronous processing. The heavy lifting of sending emails was offloaded from the main request thread.

22.2. Scenario B: Fault Tolerance and Decoupling Test

Objective: Ensure that orders are not lost if the Notification Service crashes (Service Unavailability).

Execution:

1. We manually terminated the Consumer script (simulating a server crash).
2. While the Consumer was down, we executed the Producer script three times consecutively.
3. We then restarted the Consumer script.

Result:

- During the downtime, the Producer script completed successfully without errors. The user experience was not affected by the backend failure.
- Upon restarting the Consumer, the script immediately retrieved all three "backlogged" messages from RabbitMQ and processed them sequentially.
- **Conclusion:** The system demonstrated high reliability. The order_queue successfully buffered the events, preventing data loss during service outages.

23. Deployment View (Textual Description)

Since the system is deployed in a local simulation environment using PHP Native Servers and Docker, the physical nodes and their execution environments are defined as follows:

23.1. Client Tier

- **Node:** Client Device (Web Browser).
- **Responsibility:** Renders the user interface and sends HTTP requests.
- **Protocol:** Communicates via HTTP/REST.

23.2. Application Tier (Microservices)

- **Node 1: API Gateway**
  - **Execution Environment:** PHP Built-in Server.
  - **Port:** 5000.
  - **Component:** gateway.php.
  - **Responsibility:** Entry point, Security Check (Token Validation), and Routing.
- **Node 2: Product Service**
  - **Execution Environment:** PHP Built-in Server.
  - **Port:** 5001.
  - **Component:** api.php.
  - **Responsibility:** Core business logic for managing product data.
- **Node 3: Notification Worker**
  - **Execution Environment:** PHP CLI (Background Process).
  - **Component:** consumer.php.
  - **Responsibility:** Asynchronous email processing.

23.3. Infrastructure Tier

- **Node 4: Message Broker**
  - **Software:** RabbitMQ.
  - **Port:** 5672 (AMQP).
  - **Responsibility:** Buffering and routing events between Product Service and Notification Worker.
- **Node 5: Database Server**
  - **Software:** MySQL / MariaDB.
  - **Port:** 3306.

- **Responsibility:** Persistent storage for the webbandocu database.

24. ATAM Analysis (Architecture Trade-off Analysis Method)

We evaluated the architecture based on two key scenarios: **Scalability (SS1)** and **Availability (AS1)**.

24.1. Comparison Matrix

Quality Attribute	Scenario Description	Monolithic Architecture (Lab 3)	Microservices Architecture (Lab 8)	Evaluation
Scalability	**Scenario SS1 (High Traffic):** During a flash sale event, traffic to the "View Product" feature increases by 500%, while the "Checkout" traffic remains normal.	**Inefficient:** To handle the load, we must scale the entire application (User, Order, Product modules together). This consumes excessive CPU/RAM for modules that are not under load.	**Highly Efficient:** We can spin up multiple instances of the Product Service (e.g., on ports 5002, 5003) independently. The Gateway balances the load. Resources are allocated precisely where needed.	**Microservices Wins**

Availability	**Scenario AS1 (Service Failure):** The third-party Email Service (SMTP) becomes unresponsive, or the email sending code crashes due to a bug.	**System Failure:** In a synchronous monolith, the user waits for the email to be sent. If it fails or times out, the entire "Place Order" transaction may roll back or hang, preventing sales.	**Fault Tolerant:** Thanks to the Event-Driven Architecture (Lab 7), the Order is placed immediately. The email task is queued in RabbitMQ. Even if the Notification Worker crashes, the order is safe, and emails are sent later.	**Microservices Wins**

25. Trade-off Analysis Statement

While the Microservices architecture demonstrates superior performance in the scenarios above, it introduces a significant **Trade-off**.

Trade-off: Operational Complexity vs. Scalability

- **The Cost:** The transition to Microservices has drastically increased the system's complexity. Instead of managing a single PHP application, we now manage:
    - Multiple independent processes (gateway.php, api.php, consumer.php).
    - Multiple network ports and inter-service communication rules.
    - Additional infrastructure components (RabbitMQ).

    - Debugging is more difficult as a single request spans multiple nodes (Distributed Tracing is required).
- **The Benefit:** In exchange for this complexity, we gain the ability to scale specific parts of the system and ensure that a failure in one module (e.g., Notifications) does not crash the entire business process (e.g., Ordering).