1. 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).

2. 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
```

3. Implementation Details

3.1. 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
        }
```

3.2. 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
```

3.3. 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
```

3.4. 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)
```

4. Dependency Injection (app.py)

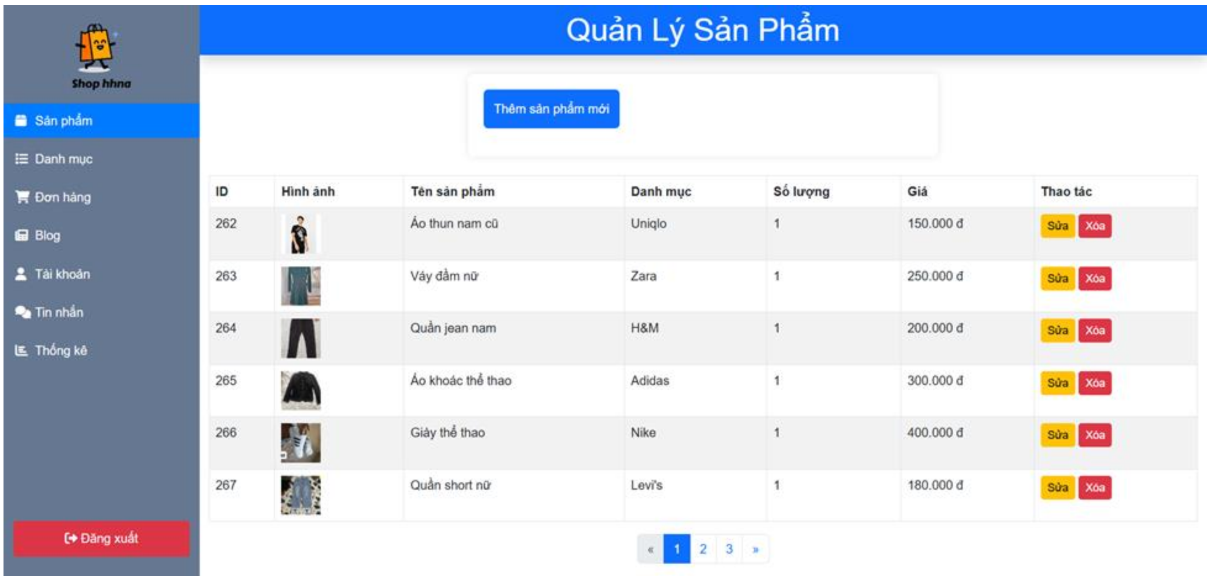
To adhere to the Inversion of Control principle, dependencies are injected at the application startup:

```
# 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
```

5. Execution & Results

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

5.2. 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.



Quản Lý Sản Phẩm

ID	Hình ảnh	Tên sản phẩm	Danh mục	Số lượng	Giá	Thao tác
262		Áo thun nam cũ	Uniqlo	1	150.000 đ	Sửa Xóa
263		Váy đầm nữ	Zara	1	250.000 đ	Sửa Xóa
264		Quần jean nam	H&M	1	200.000 đ	Sửa Xóa
265		Áo khoác thể thao	Adidas	1	300.000 đ	Sửa Xóa
266		Giày thể thao	Nike	1	400.000 đ	Sửa Xóa
267		Quần short nữ	Levi's	1	180.000 đ	Sửa Xóa