Lab 3: Layered Architecture Implementation (CRUD)

This lab focuses on the practical implementation of the **Product Management** feature using the **Layered Architecture** designed in Lab 2. We will set up the project structure and code the Create, Read, Update, and Delete (CRUD) operations for a Product, demonstrating the strict flow of control between the layers.

Objectives

1. Set up a project with distinct packages/modules for the three architectural layers (Presentation, Business Logic, Persistence).
2. Implement the Product entity and the core CRUD operations.
3. Ensure strict dependency flow: **Controller $\rightarrow$ Service $\rightarrow$ Repository**.

Technology & Tool Installation

We will use **Python** with the **Flask** framework for simplicity, and an **in-memory list** to simulate the database in the persistence layer.

Tool	**Purpose**	**Installation/Setup Guide**
Python 3.x	Core programming language.	Download and install Python from the official website.
Flask	Lightweight web framework for building the Presentation Layer (Controller/API).	Open your terminal and run: pip install Flask
IDE/Code Editor	Writing and running code (e.g., VS Code, PyCharm).	Install your preferred editor.

Activity Practice 1: Project Setup and Model Definition

Goal: Create the basic folder structure and define the data model.

Step-by-Step Instructions & Coding Guide

1. **Create Project Directory:**

Bash

```
mkdir shopsphere_layered

cd shopsphere_layered

# Create the virtual environment (recommended)

python -m venv venv

source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install Flask

pip install Flask
```

2. **Create Layer Folders/Modules:** Create the necessary subdirectories to represent the layers and the core application files.

Bash

```
mkdir presentation business_logic persistence

touch app.py  # Main entry point
```

3. **Define the Product Model:** This defines the basic structure of the data object passed between layers.

File: business_logic/models.py

Python

```
# Defines the core data structure used throughout the application

class Product:

    def __init__(self, product_id, name, price, stock):

        self.id = product_id

        self.name = name

        self.price = price

        self.stock = stock


    def to_dict(self):

        # Helper function for JSON serialization in the Presentation Layer
```

```
    return {
        "id": self.id,
        "name": self.name,
        "price": self.price,
        "stock": self.stock
    }
```

Activity Practice 2: Persistence Layer (Repository)

Goal: Implement the component that interacts directly with the data store (simulated by an in-memory dictionary).

Step-by-Step Instructions & Coding Guide

1. **Implement the Repository:** Create a class that handles data access. This layer is *unaware* of HTTP or business rules.

File: persistence/product_repository.py

Python

```
from business_logic.models import Product


# Simulates the database storage (Data Layer)
product_db = {}
next_id = 1


class ProductRepository:
    """Layer 3: Handles basic CRUD operations directly on the data store."""


    def create(self, name, price, stock):
        global next_id
        product_id = str(next_id)
```

```
    new_product = Product(product_id, name, price, stock)
    product_db[product_id] = new_product
    next_id += 1
    return new_product

def find_all(self):
    # Returns a list of Product objects
    return list(product_db.values())

def find_by_id(self, product_id):
    return product_db.get(product_id)

def update(self, product_id, name, price, stock):
    if product_id in product_db:
        product = product_db[product_id]
        product.name = name
        product.price = price
        product.stock = stock
        return product
    return None

def delete(self, product_id):
    if product_id in product_db:
        del product_db[product_id]
        return True
    return False
```

Activity Practice 3: Business Logic Layer (Service)

Goal: Implement the component that encapsulates the business rules and orchestrates calls to the Persistence Layer.

Step-by-Step Instructions & Coding Guide

1. **Implement the Service:** Create a class that provides the core business functions, using the Repository to access data.

File: business_logic/product_service.py

Python

```
from persistence.product_repository import ProductRepository

# The Service layer requires (needs) the Repository layer


class ProductService:
    """Layer 2: Handles business rules, validation, and transaction logic."""


    def __init__(self):
        # Instantiate the Repository to use its methods
        self.repo = ProductRepository()


    def create_product(self, name, price, stock):
        # Example Business Rule: Product name must be non-empty and price must be positive
        if not name or price <= 0:
            raise ValueError("Invalid product data: Name cannot be empty and price must be positive.")


        # Call the Persistence Layer (Strict downward flow)
        return self.repo.create(name, price, stock)
```

```
    def get_all_products(self):
        return self.repo.find_all()

    def get_product(self, product_id):
        product = self.repo.find_by_id(product_id)
        if not product:
            # Handle business case: Product not found
            raise ValueError(f"Product with ID {product_id} not found.")
        return product

    # Add methods for update and delete, applying relevant business rules (e.g., stock cannot be negative)
```

Activity Practice 4: Presentation Layer (Controller/API)

Goal: Implement the component that handles HTTP requests, calls the Service Layer, and formats the response.

Step-by-Step Instructions & Coding Guide

1. **Implement the Controller (Flask App):** This layer requires the Service Layer.

File: app.py

Python

```
from flask import Flask, request, jsonify
from business_logic.product_service import ProductService
# The Controller layer requires (needs) the Service layer

app = Flask(__name__)
product_service = ProductService()
```

```
@app.route('/api/products', methods=['POST'])
def create_product():
    # Layer 1: Handles request input and delegates to Layer 2
    data = request.json
    try:
        product = product_service.create_product(
            data.get('name'), data.get('price'), data.get('stock')
        )
        # Layer 1: Formats response for client
        return jsonify(product.to_dict()), 201
    except ValueError as e:
        # Layer 1: Handles exceptions raised by Layer 2 and formats an error response
        return jsonify({"error": str(e)}), 400

@app.route('/api/products', methods=['GET'])
def get_products():
    # Layer 1: Delegates request to Layer 2
    products = product_service.get_all_products()
    # Layer 1: Converts list of Product objects to list of dicts for JSON
    return jsonify([p.to_dict() for p in products]), 200

@app.route('/api/products/<product_id>', methods=['GET'])
def get_product(product_id):
    try:
        product = product_service.get_product(product_id)
```

```
        return jsonify(product.to_dict()), 200
    except ValueError as e:
        return jsonify({"error": str(e)}), 404 # 404 for Not Found


if __name__ == '__main__':
    app.run(debug=True)
```

2. **Run and Test:**
    - **Action:** Run the application from your terminal: python app.py
    - **Action:** Use a tool like **Postman** or curl to test the endpoints:
        - **Create Product (POST):** POST http://127.0.0.1:5000/api/products with JSON body: {"name": "Laptop Pro", "price": 1200.00, "stock": 5}
        - **Get All Products (GET):** GET http://127.0.0.1:5000/api/products

---

The system now demonstrates the **Layered Architecture**, where each layer is responsible for its domain, and control flows strictly downward.