Slide Deck: Lecture 3 - Layered Monolith Implementation (CRUD)

1. Title Slide: From Design to Code

- **Title:** Software Architecture: Lecture 3
- **Subtitle:** Layered Monolith Implementation (CRUD)
- **Focus:** Translating the Logical View (Layer 2) into runnable Python code for **ShopSphere**.

---

2. Recap: The Layered Contract

- **Lecture 2 Goal:** We designed the **Layered Monolith** for the Product Catalog.
- **The Flow: Controller** $\rightarrow$ **Service** $\rightarrow$ **Repository**.
- **The Rule:** Each layer **must only import and call** the layer immediately below it.
- **Goal Today:** Implement this structure using **Python and Flask** and the **CRUD** (Create, Read, Update, Delete) operations.

---

3. Knowledge Base: Separating Code by Concern

- **Concept:** In a Monolith, we use **directory structure** and **Python packages** to physically enforce the logical layer separation.
- **Why?** It ensures developers don't accidentally violate the dependency rule (e.g., calling the Repository directly from the Controller).
- **Project Structure:** We'll create three distinct folders to host our components.

---

4. Activity 1: Project Setup and Dependencies

- **Goal:** Create the physical directories and install the framework.
- **Installation Steps:**
    1. mkdir shopsphere_monolith
    2. cd shopsphere_monolith
    3. python -m venv venv
    4. source venv/bin/activate

5. pip install Flask

- **Directory Structure:**
- shopsphere_monolith/
- ├── app.py (Main Entry Point)
- ├── presentation/ (Controllers)
- ├── business_logic/ (Services & Models)
- └── persistence/ (Repositories)

---

5. Knowledge Base: The Product Model

- **Concept:** The **Model** (e.g., a Product class) is the core data structure passed *between* the layers. It lives in the Business Logic layer because it represents the domain concept.
- **Implementation:** We define a simple class with attributes and a helper method for JSON serialization.

---

6. Code Step 1: Defining the Model

- **File: business_logic/models.py**

Python

```
class Product:
    def __init__(self, id, name, price, stock):
        self.id = id
        self.name = name
        self.price = price
        self.stock = stock

    def to_dict(self):
        # Used by the Presentation layer for JSON response
```

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

---

7. Knowledge Base: The Persistence Layer (Repository)

- **Role:** To isolate the application from the underlying data storage mechanism.
- **Data Store:** For simplicity, we use a Python **in-memory dictionary** (product_db) instead of a full database. The Repository hides this detail.
- **Core Function:** Implement the low-level **CRUD** methods (create, find_by_id, update, delete).

---

8. Code Step 2: Implementing the Repository

- **File: persistence/product_repository.py**

Python

```
from business_logic.models import Product

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

class ProductRepository:
    def find_all(self):
        return list(product_db.values())
```

```
    def create(self, name, price, stock):
        global next_id
        product_id = next_id
        new_product = Product(product_id, name, price, stock)
        product_db[product_id] = new_product
        next_id += 1
        return new_product

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

---

9. Knowledge Base: The Business Logic Layer (Service)

- **Role:** Contains the **"Why"** of the application—the business rules.
- **Constraint: MUST** import and use the ProductRepository.
- **Value-Add:** Before calling the Repository, the Service should perform validation or logic (e.g., checking for valid inputs, calculating taxes, etc.).

---

10. Code Step 3: Implementing the Service

- **File: business_logic/product_service.py**

Python

```
# ONLY imports from the persistence layer (Layer N-1)
from persistence.product_repository import ProductRepository

class ProductService:
    def __init__(self):
        self.repo = ProductRepository()
```

```
    def create_product(self, name, price, stock):
        # Business Rule 1: Price must be positive
        if price <= 0:
            raise ValueError("Product price must be greater than zero.")
        # Business Rule 2: Stock must be non-negative
        if stock < 0:
            stock = 0

        # Strict Downward Call
        return self.repo.create(name, price, stock)

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

---

11. Knowledge Base: The Presentation Layer (Controller)

- **Role:** The application's interface to the outside world (HTTP requests).
- **Constraint: MUST** import and use the ProductService. **MUST NOT** import ProductRepository.
- **Tasks:** Parse request data, call the Service, format the HTTP response (JSON), and handle HTTP status codes (400, 200, 201, 404).

---

12. Code Step 4: The Main Controller (app.py)

- **File: app.py**

Python

```
from flask import Flask, request, jsonify

# ONLY imports from the business_logic layer (Layer N-1)
```

```
from business_logic.product_service import ProductService

app = Flask(__name__)
product_service = ProductService() # Instantiate the service

@app.route('/api/products', methods=['POST'])
def add_product_api():
    data = request.json
    try:
        # 1. Call the Service Layer
        new_product = product_service.create_product(
            data.get('name'), data.get('price'), data.get('stock')
        )
        # 2. Format Response
        return jsonify(new_product.to_dict()), 201
    except ValueError as e:
        # Handle business logic exceptions
        return jsonify({"error": str(e)}), 400

# Add main execution block...
```

---

13. Activity 2: Proof of Concept - Dependency Enforcement

- **Goal:** Visually confirm the strict rule is enforced by the code's imports.
- **Demonstration:**
    - Look inside product_service.py: It imports product_repository.py. (Correct: Layer 2 $\rightarrow$ Layer 3)

- Look inside app.py: It imports product_service.py. (Correct: Layer 1 $\rightarrow$ Layer 2)
- **Crucial Check:** If a developer tries to add from persistence.product_repository import * to app.py, it breaks the architecture and leads to coupling!

---

14. Code Step 5: The Read Endpoint (R in CRUD)

- **File: app.py (Add to existing routes)**

Python

```
@app.route('/api/products', methods=['GET'])
def get_products_api():
    # Call the Service, which calls the Repository
    products = product_service.get_all_products()
    # Format Response
    return jsonify([p.to_dict() for p in products]), 200
```

---

15. Activity 3: Practical Implementation Steps

- **Goal:** Run the application and test the implemented C and R operations.
- **Action Steps:**
    1. Complete all code files as outlined.
    2. Run the application: python app.py
    3. Verify the server is running on http://127.0.0.1:5000.

---

16. Verification (Hands-On): Testing the Flow

- **Tool:** Use Postman or cURL.
- **Test 1: Create Success (C):**
    - **Method:** POST
    - **URL:** http://127.0.0.1:5000/api/products

    - **Body:** {"name": "Laptop X1", "price": 1000.00, "stock": 5}
    - **Expected:** HTTP **201 Created**.
- **Test 2: Create Failure (Business Rule):**
    - **Method:** POST
    - **URL:** http://127.0.0.1:5000/api/products
    - **Body:** {"name": "Freebie", "price": -5.00, "stock": 10}
    - **Expected:** HTTP **400 Bad Request** (Handled by the Service Layer).

---

17. Verification (Cont.): Testing the Read Operation (R)

- **Test 3: Read All (R):**
    - **Method:** GET
    - **URL:** http://127.0.0.1:5000/api/products
    - **Expected:** HTTP **200 OK** with a JSON array containing "Laptop X1".

---

18. Architectural Review: Monolith Pros and Cons

- **Pro:** We achieved fast setup and clear **Separation of Concerns (SoC)**. If we switch from in-memory data to MySQL, only the **Persistence Layer** changes.
- **Con: Scaling:** If we need 10x capacity for the product list, we must deploy 10 copies of the **entire** system (including the User Admin module). This validates the need to move away from this pattern later.

---

19. Summary of Deliverables for Submission

- **Document 1:** Screenshot of the final project directory structure.
- **Document 2:** Code snippet of the create_product method in the **Service Layer** showing the business rule (price validation).
- **Document 3:** Output logs/screenshots showing the successful **POST (201)** request and the failed **POST (400)** request due to the enforced business rule.

---

20. Q&A and Next Steps

- **Questions?** (Review error handling flow from Service to Controller).
- **Pre-work:** Research **Microservice Decomposition**—how to identify boundaries based on business functions.
- **Next Lecture: Lecture 4: Microservices Decomposition & Context**. We break the Monolith into independent services, driven by the Scalability ASR.