Software Architecture: Lecture 3
3

Layered Monolith Implementation (CRUD)
(CRUD)

Welcome to Lecture 3! Today, we bridge the gap between theoretical design and practical implementation. We'll be translating the logical view of our Layered Monolith (Layer 2) into runnable Python code for our e-commerce platform, ShopSphere. Our focus will be on understanding how to structure a Python/Flask application to enforce architectural principles, specifically for **CRUD (Create, Read, Update, Delete)** operations.

Recap: The Layered Contract

In Lecture 2, we meticulously designed the Layered Monolith for ShopSphere's Product Catalog. This design established a clear, strict contract between layers. Let's revisit the core principles:

Controller

The outermost layer, responsible for handling incoming requests and returning responses.

Service

Contains the core business logic and orchestrates operations across multiple repositories if needed.

Repository

Abstracts the data storage mechanism, providing an interface for data access.

The fundamental rule of this contract is paramount: **"Each layer must only import and call the layer immediately below it."** This strict dependency rule is critical for maintaining separation of concerns and ensuring the maintainability and testability of our system. Today, we will implement this structure using Python and Flask, focusing on the foundational CRUD operations that underpin most applications.

Knowledge Base: Separating Code by Concern

In a monolith, while all code lives within a single deployable unit, we must still physically enforce the logical layer separation. This is achieved through a well-organized **directory structure** and the judicious use of **Python packages**.

> **Why is this crucial?** It prevents developers from accidentally violating the architectural dependency rule. For instance, a developer should not be able to directly call a Repository method from a Controller without going through the Service layer. This structural enforcement is a design pattern in itself, promoting discipline and reducing technical debt.

For ShopSphere, we'll establish three distinct folders, each acting as a Python package, to host our components:

`presentation/`

Houses our Flask Controllers, handling HTTP requests and responses.

`business_logic/`

Contains our Services and core Domain Models, embodying the application's business rules.

`persistence/`

Encapsulates data access logic, such as Repositories, isolating the application from the database.

This clear separation provides a robust framework for development, making it easier to onboard new team members and maintain code quality over time.

Activity 1: Project Setup and Dependencies

Our first step is to establish the foundational environment for our ShopSphere monolith. This involves creating the necessary physical directories and installing our core framework, Flask.

Installation Steps:

```
1. mkdir shopsphere_monolith2. cd shopsphere_monolith3. python -m venv venv4. source venv/bin/activate  # On Windows, use `venv\Scripts\activate`5. pip install Flask
```

These commands will create our project root, set up a Python virtual environment to manage dependencies, and install Flask, the micro-framework we'll use to build our web application.

Envisioning the Directory Structure:

After setup, our project will adhere to the following structure, which physically mirrors our layered design:

```
shopsphere_monolith/ ├── app.py            (Main Entry Point) ├── presentation/      (Controllers)│   └── __init__.py ├── __init__.py ├── business_logic/    (Services & Models)│   └── __init__.py└── persistence/       (Repositories)   └── __init__.py └── __init__.py
```

The `__init__.py` files mark these folders as Python packages, allowing us to import modules across different layers cleanly. This setup is not just for organization; it's a critical component of enforcing our architectural rules.

Knowledge Base: The Product Model

At the heart of any e-commerce application lies its core data entities. In ShopSphere, the **Product Model** serves as the fundamental data structure that flows between our architectural layers.

Core Data Structure

The Product class defines what a "product" means within our application: its attributes (ID, name, price, stock) and its behavior.

Domain Concept

Because it represents a core domain concept, the Model logically resides within the **Business Logic layer**. This ensures that the definition of a product is central to our business rules.

Cross-Layer Communication

Instances of the Product Model will be passed from the Persistence layer up to the Service layer, and then to the Presentation layer, acting as a uniform contract across the entire system.

In Python, we'll implement this as a simple class. Additionally, we'll include a helper method for JSON serialization, which is crucial for sending product data efficiently to client applications via our API. This serialization method ensures that our internal data representation can be easily consumed by external interfaces.

Code Step 1: Defining the Model

Let's implement our `Product` model within the `business_logic` package. This class will serve as the blueprint for all product data within ShopSphere, defining its properties and how it can be represented.

File: `business_logic/models.py`

```
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):        # This method is used by the Presentation layer        # to convert the Product object into a dictionary        # suitable for JSON
responses.        return {            "id": self.id,            "name": self.name,            "price": self.price,            "stock": self.stock        }    def __repr__(self):
# A helpful representation for debugging        return f"Product(id={self.id}, name='{self.name}', price={self.price}, stock={self.stock})"
```

Key elements:

- `__init__(...)`: The constructor takes essential product attributes: `id`, `name`, `price`, and `stock`. These are direct representations of a product's core information.
- `to_dict()`: This method is vital for the Presentation Layer. It converts the `Product` object into a Python dictionary, which Flask's `jsonify` function can then seamlessly transform into a JSON response for our API clients. This decouples the internal object representation from the external API format.
- **Placement:** By placing `models.py` in the `business_logic` directory, we ensure that our core domain entities are defined alongside the business rules that govern them, reinforcing the architectural separation of concerns.

Knowledge Base: The Persistence Layer (Repository)

The **Persistence Layer**, typically implemented via the **Repository Pattern**, plays a crucial role in abstracting our application from the underlying data storage mechanism.

1

Isolation

The primary goal of this layer is to isolate the rest of the application from the specifics of how data is stored, retrieved, updated, or deleted.

2

Data Store Agnostic

For simplicity, we'll start with an **in-memory Python dictionary** (`product_db`) instead of a full-fledged database. This choice allows us to focus on the pattern, not database-specific syntax. Crucially, the Repository hides this implementation detail entirely. If we later switch to PostgreSQL, MongoDB, or a cloud database, only the Persistence Layer needs modification.

3

CRUD Operations

The core function of the Repository is to provide a standardized interface for performing basic CRUD (Create, Read, Update, Delete) operations on our domain entities (e.g., `Product`).

This layer serves as a gateway to our data, ensuring that higher layers (Business Logic, Presentation) interact with data in a consistent, technology-agnostic manner.

Code Step 2: Implementing the Repository

Now, let's create our `ProductRepository` within the `persistence` package. This class will manage our product data, currently simulated by an in-memory dictionary.

File: `persistence/product_repository.py`

```
from business_logic.models import Product# Simulates the Data Layer (in-memory database)product_db = {}next_id = 1class ProductRepository:    def find_all(self):        # Returns a list of all products in our simulated
database.        return list(product_db.values())    def create(self, name, price, stock):        global next_id # We use global for simplicity in this example        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):        # Retrieves a product by its ID, or None if not found.        return
product_db.get(product_id)    # Future implementations would include update and delete methods:    # def update(self, product_id, name=None, price=None, stock=None):    #     product = self.find_by_id(product_id)    #
if product:    #         if name is not None: product.name = name    #         if price is not None: product.price = price    #         if stock is not None: product.stock = stock    #     return product    # def delete(self, product_id):    #
if product_id in product_db:    #         del product_db[product_id]    #         return True    #     return False
```

Knowledge Base: The Business Logic Layer (Service)

The **Business Logic Layer**, typically embodied by our **Service classes**, is the brain of our application. It contains the "Why" – the specific rules, validations, and orchestrations that define how our business operates.

Core Business Rules

This layer enforces the domain-specific logic. For example, ensuring a product's price is positive or managing stock levels.

Strict Downward Calls

A Service **MUST** import and utilize the `ProductRepository`. It acts as the direct interface to data operations, but does so by applying business context.

Value-Add and Validation

Before delegating to the Repository, the Service performs critical validation and applies business logic. This might include checking input validity, performing calculations (e.g., tax, discounts), or coordinating operations across multiple repositories.

This layer shields the Presentation Layer from direct interaction with the Persistence Layer, ensuring that business rules are consistently applied, regardless of how or where a request originates. It's a critical checkpoint for maintaining data integrity and application correctness.

Code Step 3: Implementing the Service

Let's define our `ProductService` within the `business_logic` package. This service will orchestrate product operations, including business rule enforcement, before interacting with the data layer.

File: `business_logic/product_service.py`

```
# ONLY imports from the persistence layer (Layer N-1), as per architectural rules.from persistence.product_repository import ProductRepositoryfrom business_logic.models import Product # Also needs the Product
modelclass ProductService:    def __init__(self):        # Instantiate the repository. This is a dependency injection point.        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 a negative stock is provided, default it to 0 instead
of raising an error.        if stock < 0:            stock = 0            print(f"Warning: Stock for '{name}' was negative, setting to 0.") # Log a warning        # Strict Downward Call: Delegate the actual creation to the repository.        #
The service adds value by applying rules *before* persisting.        return self.repo.create(name, price, stock)    def get_all_products(self):        # Simple delegation: retrieve all products from the repository.        # In a more
complex scenario, there might be caching or filtering logic here.        return self.repo.find_all()    # Future implementations would include update and delete methods, with their own business rules:    # def
update_product(self, product_id, name=None, price=None, stock=None):    #     if price is not None and price <= 0:    #         raise ValueError("Product price must be greater than zero.")    #     if stock is not None and stock
< 0:    #         stock = 0    #     return self.repo.update(product_id, name, price, stock)    # def delete_product(self, product_id):    #     # Example: Add a business rule that prevents deletion of products in active orders.    #
# if self.order_service.is_product_in_active_orders(product_id):    #     #     raise BusinessRuleViolationError("Cannot delete product with active orders.")    #     return self.repo.delete(product_id)
```