Slide Deck: Lecture 2 - Layered Architecture Pattern (Logical View)

1. Title Slide: Building the Monolith

- **Title:** Software Architecture: Lecture 2
- **Subtitle:** The Layered Architecture Pattern (Logical View)
- **Focus:** Designing the internal structure of the **ShopSphere** Monolith.

---

2. Recap: The Architectural Drivers

- **Last Week:** We defined the **ASRs** (Scalability, Fault Isolation, Security) and the **Use Case View** (Make Purchase).
- **Challenge:** We need a simple, organized structure to start building features like the Product Catalog while meeting initial non-functional goals.
- **Solution:** The **Layered Architecture**—the most common pattern for starting a monolithic application.

---

3. Knowledge Base: The Layered Pattern

- **Description:** An architectural pattern where components are organized into horizontal layers, each performing a specific role within the system.
- **Analogy:** Building floors in an office building. Each floor has a distinct function (lobby, operations, executive offices) and strict access rules.
- **Key Advantage: Separation of Concerns (SoC)**. Components in one layer only focus on their specialized tasks (e.g., the database layer knows nothing about web requests).

---

4. The Crucial Rule: Strict Downward Dependency

- **The Constraint:** A component in layer N can **only** call services or components in layer N-1 (the layer immediately below it).
- **The Prohibited Action:** A component in Layer 3 (Persistence) **must never** call a component in Layer 1 (Presentation).
- **Benefit of Strictness: Maintainability** and **Testability**. Changes in the Presentation layer don't break the Database layer, and vice-versa.

---

5. The Four Standard Layers of a Monolith

Layer	Responsibility	ShopSphere Example
1. Presentation (UI/Web)	Handles user requests (HTTP/APIs), authentication, UI rendering.	**ProductController**
2. Business Logic (Service/Domain)	Executes core business rules, validation, and transaction management.	**ProductService**
3. Persistence (Data Access)	Maps objects to database records (ORM/DAO), handles CRUD execution.	**ProductRepository**
4. Data (Database)	The physical storage system (e.g., PostgreSQL, SQLite).	**products Table**

---

6. Knowledge Base: Request Flow in Layered Architecture

- **Process View:** Visualizing how data moves through the layers.
- **Scenario:** Customer clicks "View Product Details."
- **The Flow:**
    1. **Browser** $\rightarrow$ **Presentation** (HTTP Request)
    2. **Presentation** $\rightarrow$ **Business Logic** (Call getProductDetails(id))
    3. **Business Logic** $\rightarrow$ **Persistence** (Call findById(id))
    4. **Persistence** $\rightarrow$ **Data** (Execute SQL)
    5. Data $\rightarrow$ **Persistence** $\rightarrow$ **Business Logic** $\rightarrow$ **Presentation** $\rightarrow$ **Browser** (Response)

---

7. Activity 1: Defining Layer Responsibilities

- **Goal:** Formalize the role of each layer for the **ShopSphere** Product Catalog feature.

- **Practical Activity:** Students define the primary function and input/output for the top three layers.

Layer	Primary Function	Input	Output
Presentation	Process API requests.	HTTP Request (JSON/Form Data)	HTTP Response (JSON/HTML)
Business Logic	Enforce business rules.	Data Model Object (e.g., Product)	Validated Data Model Object
Persistence	Database mapping.	SQL/ORM Command	Database Record/Tuple

---

8. Knowledge Base: The Logical View (UML Component)

- **Purpose:** To show the **static structure** of the system.
- **Artifact:** The **UML Component Diagram**.
- **Component:** A self-contained, replaceable, and encapsulated part of the system that provides services through interfaces (e.g., a class, module, or service).

---

9. Component Modeling: Interfaces

- **Interfaces** define the contract between components. They show *what* services are provided and *what* services are required.
- **Provided Interface (Lollipop):** A service offered by the component (the public methods).
- **Required Interface (Socket):** A service the component needs from another component to function.
- **Connection:** The Socket of the Consumer plugs into the Lollipop of the Producer.

---

10. Activity 2: Component Identification (Product Catalog)

- **Goal:** Break down the Product Catalog feature into concrete components that live in each layer.

Layer	Component Name	Role/Provided Interface Example
Presentation	**ProductController**	Receives GET /products/{id}.
Business Logic	**ProductService**	**Provides:** IProductService (Method: getProductDetails(id)).
Persistence	**ProductRepository**	**Provides:** IProductRepository (Method: findById(id)).

---

11. Proof of Concept: Component Dependencies

- **Statement:** The **ProductController requires** the **ProductService**.
- **Statement:** The **ProductService requires** the **ProductRepository**.
- **Architectural Principle:** This chaining of required interfaces enforces the strict downward dependency rule designed for maintainability.

---

12. Practical Activity (Hands-On): Drawing Components

- **Tool:** draw.io (Diagrams.net).
- **Steps:**
    1. Draw three large, stacked rectangles labeled: **Presentation**, **Business Logic**, **Persistence**.
    2. Inside each, draw the corresponding **UML Component** shape (the rectangle with two smaller squares).
    3. Label the components: ProductController, ProductService, ProductRepository.

---

13. Practical Activity (Cont.): Modeling Interfaces

- **Step 4 (Service Provided):** Attach a **Lollipop (provided)** to the ProductService and label it IProductService.
- **Step 5 (Controller Required):** Attach a **Socket (required)** to the ProductController.
- **Step 6 (Connecting):** Draw a line from the Controller's Socket to the Service's Lollipop. This shows the dependency flow.

- **Step 7 (Repository Provided):** Repeat Steps 4-6 for the connection between the ProductService (Consumer/Socket) and the ProductRepository (Producer/Lollipop).

---

14. Visualization: The Complete Logical View

---

15. Architectural Trade-Offs of the Layered Pattern

- **Pro (Why we chose it first):**
    - **Simplicity:** Easy to understand and develop rapidly for small teams.
    - **Testability:** Each layer can be tested in isolation (mocking the layer below).
- **Con (Why we may abandon it later):**
    - **Scaling Inefficiency:** If only the Catalog needs capacity, you must scale the entire monolithic application (including the Admin UI and Checkout logic).
    - **Deployment Coupling:** A change in the Presentation Layer requires redeploying the whole system.

---

16. Analyzing ASRs: Layered vs. Microservices (Initial)

- **ASR 1 (Scalability):**
    - **Layered:** Fails easily. Scaling is inefficient and expensive.
    - **Microservices (Future):** Excels. Allows independent scaling of high-demand services (Product Catalog).
- **ASR 2 (Fault Isolation):**
    - **Layered:** Fails. A bug in the business logic can crash the entire application instance.
- **Conclusion:** The Layered Monolith is a good starting point for **rapid development**, but it's fundamentally weak against our critical ASRs (Scalability and Availability).

---

17. Project Integration: Linking to Lecture 3

- **Lecture 3 Focus:** We will now **implement** this three-layer design (Controller → Service → Repository) using Flask and Python.
- **Goal:** Use folder structure and file imports to *physically* enforce the strict dependency rule modeled in this lecture.

---

18. Self-Correction: Common Modeling Errors

- **Error 1:** Drawing the arrow from the Repository back up to the Controller. **(Violation!)** Remember: Dependencies must only point downward.
- **Error 2:** Missing the interface notation (Lollipop/Socket). Without it, the diagram only shows *usage*, not the *contract* that enables interchangeability.
- **Error 3:** Mixing up the system boundary (Lab 1) with the logical layers (Lab 2).

---

19. Summary of Deliverables for Submission

- **Document 1:** Table defining the purpose and output artifacts for the Presentation, Business Logic, and Persistence layers.
- **Document 2:** The final **UML Component Diagram** (from draw.io) for the **Product Catalog**, clearly showing the components, interfaces (lollipop/socket), and the strict downward dependency flow.

---

20. Q&A and Next Steps

- **Questions?** (Review component relationships and dependency rules).
- **Pre-work:** Install **Python** and **Flask** for the next hands-on coding lecture.
- **Next Lecture: Lecture 3: Layered Monolith Implementation (CRUD)**. We build the architecture we designed today.