Slide Deck: Lecture 4 - Microservices Decomposition & Context

1. Title Slide: Breaking the Monolith

- **Title:** Software Architecture: Lecture 4
- **Subtitle:** Microservices Decomposition & Context
- **Focus:** Transitioning from the Layered Monolith (Lecture 3) to an independent, distributed **Microservices Architecture**.

---

2. Recap: Monolith's Weakness vs. ASRs

- **Monolith Problem:** The layered approach (Lecture 3) fails our core **Scalability ASR**. If the Product Catalog sees heavy load, we must scale the entire monolithic application, wasting resources on underutilized components (like Admin Tools).
- **Solution Requirement:** We need **Service Independence** to allow different parts of the system to be developed, deployed, and scaled autonomously.
- **Architectural Choice: Microservices Architecture**.

---

3. Knowledge Base: Microservices Definition

- **Definition:** An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms (usually HTTP/REST or messaging).
- **Key Principles:**
    1. **Service Independence:** Each service can be deployed, scaled, and fail independently.
    2. **Data Ownership:** Each service owns its database. No cross-service database access.
    3. **Decomposition:** Organized around **business capabilities**.

---

4. Knowledge Base: Decomposition by Business Capability

- **Principle:** Instead of structuring code by technical concerns (Controller, Service, Repository), services are structured around things the business *does* (Catalog Management, Order Fulfillment, User Authentication).
- **Benefit: High Cohesion** (everything related to "Product" is in the Product Service) and **Low Coupling** (changes to the Cart Service don't affect the Order Service).

---

5. Activity 1: Identifying Core Capabilities (ShopSphere)

- **Goal:** Define the independent domains from the monolithic structure.
- **Practical Activity:** Break down ShopSphere's functions:
    1. **Product/Catalog:** Viewing, searching, managing details.
    2. **User/Account:** Login, registration, profile management.
    3. **Order/Fulfillment:** Checkout, tracking, history.
    4. **Inventory:** Managing stock levels and availability.
    5. **Shopping Cart:** Session state and item aggregation.

---

6. Activity 2: Defining the Microservices

- **Goal:** Map each capability to an independent service.

Business Capability	**Proposed Microservice**	**Owned Entities/Data**
Catalog Management	**Product Service**	Product Details, Price, Description.
User Management	**User Service**	User Profile, Authentication Tokens.
Fulfillment	**Order Service**	Order Header, Line Items, Status.
Stock Control	**Inventory Service**	Stock Quantity, Warehouse Location.

---

7. Knowledge Base: Service Contracts (APIs)

- **Concept:** A service contract is the public, stable interface a service exposes to the outside world or to other services. For web services, this is typically a **RESTful API specification**.

- **Constraint:** The contract **must not** expose the service's internal persistence details (e.g., specific database technology).
- **Importance:** Allows clients (other services) to integrate without knowing the service's internal implementation details.

---

8. Practical Activity: Defining the Product Service Contract

- **Goal:** Document the RESTful interface for the **Product Service** (Producer).

Endpoint	**HTTP Method**	**Description**	**Data Returned**
/api/products	**GET**	Retrieves a list of products (e.g., search or browse).	List of Product Summary objects.
/api/products/{id}	**GET**	Retrieves full details for one product.	Full Product object (includes price, description).
/api/products/{id}/price	**GET**	Retrieves only the price (useful for Cart Service).	{ "price": 1200.00 }

---

9. Knowledge Base: Inter-Service Communication

- Microservices must talk to each other to fulfill a request.
- **Type 1: Synchronous Communication (Request/Response):**
  - *Mechanism:* HTTP/REST (like the Monolith).
  - *Use Case:* Actions that require an immediate response (e.g., Cart Service requesting the current Price from the Product Service).
  - *Drawback:* High coupling; if the called service is down, the calling service blocks and fails.
- **Type 2: Asynchronous Communication (Events/Messaging):**
  - *Mechanism:* Message Brokers (RabbitMQ, Kafka).

- *Use Case:* Actions that can be processed later (e.g., Order Service notifying the Notification Service that an order was placed).
- *Benefit:* High decoupling and fault tolerance.

---

10. Activity 3: Designing Communication Strategy

- **Goal:** Decide which communication type suits which interaction based on the **Fault Isolation ASR (ASR 2)**.

Interaction	Service Flow	Type	Rationale
Checkout Price Check	Cart → Product	**Synchronous**	Must fail immediately if price is unavailable or wrong.
Order Confirmation	Order → Notification	**Asynchronous**	Email delivery must not block the core business transaction (order placement).
Inventory Update	Order → Inventory	**Asynchronous**	Stock deduction can happen after the order confirmation, increasing resilience.

---

11. Knowledge Base: The C4 Model

- **Purpose:** A standardized way to visually document software architecture at different levels of detail, helping to communicate complexity to different stakeholders.
- **Level 1: System Context Diagram:** The highest level. Shows the entire software system as a whole and how it relates to users and other external systems.
- **Elements:** System (Box), User (Stick figure), External System (Box).

---

12. Activity 4: Modeling the System Context (C4 L1)

- **Goal:** Visualize the ShopSphere system boundary and its external interactions.
- **Practical Activity:** Sketching the high-level map.
- **System: [System] ShopSphere E-commerce Platform** (The main focus).
- **Actors:**

    1. **[Person] Web Customer** (Browses and places orders).
    2. **[Person] Administrator** (Manages catalog and users).

- **External Systems:**
    1. **[System] Payment Gateway** (Receives payment submissions).
    2. **[System] Email Service** (Sends confirmations and promotions).

---

13. Practical Activity (Hands-On): Drawing the C4 L1 Diagram

- **Tool:** draw.io (Diagrams.net) (Use the C4 Model Stencil).
- **Steps:**
    1. Draw the main **ShopSphere** system box in the center.
    2. Place the **Web Customer** and **Administrator** outside the box.
    3. Draw directional arrows:
        - Customer → ShopSphere (Uses application).
        - ShopSphere → Payment Gateway (Submits payment synchronously).
        - ShopSphere → Email Service (Sends notifications asynchronously).
    4. Label the communication protocols (e.g., HTTP/S, SMTP/API).

---

14. Visualization: The C4 L1 System Context

---

15. Architectural Trade-Off: Microservices vs. Monolith

- **Microservices Gain: Scalability** and **Fault Isolation** (meeting ASRs). A crash in the Inventory Service doesn't stop the User Service.
- **Microservices Cost (The New Problem): Complexity**. We introduced distributed transactions, networking overhead, service discovery, and the need for an API Gateway.
- **Trade-off Statement:** We trade development simplicity for operational complexity to achieve critical non-functional requirements.

---

16. Project Integration: Linking to Lecture 5

- **Lecture 5 Focus:** Implementing the **Product Service**, the simplest independent component, following the contract defined today.
- **Key Concept:** We must ensure the Product Service is *truly* independent, owning its own database and running in isolation.

---

17. Self-Correction: Decomposition Traps

- **Trap 1: The God Service:** Creating a "Core" or "Common" service that everything else depends on. **(Violation!)** This introduces tight coupling.
- **Trap 2: Distributed Monolith:** Breaking the code into separate repos but having services share the same database. **(Violation!)** This prevents independent scaling and deployment.
- **Solution:** Focus on the **business entity** (Product, Order, User) as the primary organizational principle.

---

18. Discussion: Data Integrity in Distributed Systems

- **Challenge:** How do we ensure inventory is reserved *before* payment is finalized, given the services are separate?
- **Monolith Solution:** A single database transaction (ACID).
- **Microservice Solution: Saga Pattern** (a sequence of local transactions, potentially using events/messages) or **Two-Phase Commit** (usually avoided). We will explore Saga concepts later.

---

19. Summary of Deliverables for Submission

- **Document 1:** The **Decomposition Table** listing the four core microservices and their data ownership.
- **Document 2:** The documented **Service Contract** (API Endpoints) for the Product Service.
- **Document 3:** The final **C4 Model (Level 1: System Context Diagram)** showing all external interactions.

20. Q&A and Next Steps

- **Questions?** (Focus on Synchronous vs. Asynchronous use cases).
- **Pre-work:** Install **Flask-SQLAlchemy** and the **Redis server** (if not already done).
- **Next Lecture: Lecture 5: Implementing an Independent Microservice**. We build the Product Service in isolation.