Software Architecture: Lecture 4

Microservices Decomposition & Context Context

Welcome to Lecture 4, where we transition from the monolithic structures discussed in Lecture 3 to the independent, distributed world of Microservices Architecture. This lecture will equip you with the foundational knowledge and practical skills needed to effectively decompose a complex application into manageable, scalable microservices. We'll explore the principles behind microservices, delve into decomposition strategies, define service contracts, and understand inter-service communication patterns.

From Monolith to Microservices: Addressing Core Challenges

The Monolith Problem

Our previous lecture highlighted the limitations of the layered monolithic approach. While simple to develop initially, monoliths often struggle with scalability, especially when different components have varying load requirements. For instance, if the Product Catalog experiences heavy traffic, the entire monolithic application must scale, leading to inefficient resource allocation for less utilized components like Admin Tools. This inherent coupling restricts independent scaling and deployment, becoming a significant bottleneck for evolving systems.

The Microservices Solution

To overcome these challenges, we need to achieve Service Independence. This architectural requirement demands that different parts of the system can be developed, deployed, and scaled autonomously. Our chosen architectural solution is Microservices Architecture, which promotes breaking down a single application into a suite of small, independent services. This approach ensures that each service can run in its own process, manage its own data, and communicate through lightweight mechanisms, typically HTTP/REST or asynchronous messaging.

Microservices Defined: Core Principles

At its core, a microservice architecture structures a single application as a collection of loosely coupled services. Each service embodies a specific business capability, operating independently, and communicating via well-defined APIs.

Service Independence

Each microservice is designed to be fully autonomous. This means it can be developed, deployed, scaled, and even fail independently without affecting the entire system. This isolation is crucial for resilience and agility.

Data Ownership

A fundamental principle: each service owns its data store. There is no sharing of databases between services. This prevents tight coupling and ensures that internal data schema changes within one service do not impact others.

Decomposition by Business Capability

Microservices are organized around business capabilities, not technical layers. Instead of a "User Interface," "Business Logic," and "Database" layer, you'd have services like "Order Fulfillment" or "User Management."

Decomposition by Business Capability

Moving away from traditional layered architectures, microservices emphasize structuring services around distinct business functions. This strategic approach yields significant benefits in system design and maintainability.

Principle: Business-Oriented Design

Instead of organizing code by technical concerns—such as a Controller, Service, and Repository layer for an entire application—microservices are structured around the specific functions a business performs. Think of services like "Product Catalog Management," "Order Fulfillment," or "User Authentication." This aligns the software architecture directly with the business domain, making services more understandable and aligned with organizational structures.

This approach ensures that all logic, data, and interfaces related to a particular business capability are encapsulated within a single service.

Key Benefits

- **High Cohesion:** All elements related to a specific business function are grouped together. For example, everything related to a "Product" (its details, pricing, inventory association) resides within the Product Service. This makes the service easier to understand, develop, and maintain.
- **Low Coupling:** Changes made within one service, such as a modification to the Cart Service, have minimal to no impact on other services like the Order Service. This reduces the ripple effect of changes across the system, enabling independent development and deployment cycles.

By focusing on business capabilities, microservices promote modularity and reduce interdependencies, paving the way for more agile development and robust systems.

Activity 1: Identifying Core Capabilities (ShopSphere)

Our first practical step in microservices decomposition is to identify the independent business domains within our hypothetical e-commerce application, **ShopSphere**. This exercise helps us break down the monolithic structure into logical, self-contained units.

Product/Catalog Management

This domain encompasses all functionalities related to displaying, searching, and managing product details. Think about attributes like product name, description, images, and categories.

User/Account Management

This capability handles user-related actions, including registration, login, authentication, and the management of user profiles and preferences.

Order/Fulfillment

This domain covers the entire order lifecycle, from processing checkout to tracking orders, managing payment status, and maintaining order history for customers.

Inventory Management

Dedicated to tracking and managing stock levels, ensuring product availability, and handling updates as items are sold or restocked.

Shopping Cart Management

Manages the session state for a customer's shopping cart, allowing users to add, remove, and aggregate items before proceeding to checkout.

By isolating these capabilities, we lay the groundwork for defining distinct microservices, each responsible for a single, well-defined business function.

Activity 2: Defining the Microservices

Now that we've identified the core business capabilities, the next step is to map each capability to an independent microservice and define the specific data it will own. This ensures data ownership and promotes true service independence.

Catalog Management	Product Service	Product Details, Price, Description, Images, Categories
User Management	User Service	User Profile, Authentication Tokens, Preferences, Shipping Addresses
Order Fulfillment	Order Service	Order Header, Line Items, Status, Payment Information Reference
Stock Control	Inventory Service	Stock Quantity, Warehouse Location, SKU, Product Availability
Shopping Cart	Cart Service	Cart Items, User Session Data, Quantities

Each service will now be responsible for its own data, preventing direct database access from other services and enforcing clear boundaries. This table forms the foundation for our microservices architecture.

Knowledge Base: Service Contracts (APIs)

In a microservices architecture, how services interact is paramount. The concept of a Service Contract defines this interaction, establishing clear boundaries and expectations.

Defining the Public Interface

A service contract is the public, stable interface that a service exposes. This interface allows other services, or external clients, to interact with it without needing to know its internal implementation details. For web services, this is most commonly expressed as a **RESTful API specification**, detailing endpoints, HTTP methods, request/response formats, and data schemas.

Crucial Constraint

The contract **must not** expose the service's internal persistence details. This means avoiding references to specific database technologies, table structures, or ORM specifics in the public API. The internal data model is an implementation detail of the service and can change without affecting its consumers, as long as the contract remains stable.

Importance: A well-defined service contract acts as a binding agreement between a service and its consumers. It enables clients to integrate seamlessly and reliably, fostering independent

Practical Activity: Defining the Product Service Contract

Let's apply the concept of service contracts by defining a RESTful interface for our **Product Service**. This service will act as a "producer" of product information, consumed by other services (like the Cart Service) and potentially by the frontend application.

/api/products	GET	Retrieves a list of products based on search criteria or for browsing.	List of Product Summary objects (e.g., id, name, thumbnail_url, basic_price)
/api/products/{id}	GET	Retrieves full details for a specific product by its unique ID.	Full Product object (includes id, name, description, full_price, images, specifications, stock_status)
/api/products/{id}/price	GET	Retrieves only the current price for a product. Useful for services like Cart Service to avoid fetching full product details just for pricing.	`{"productId": "{id}", "currency": "USD", "amount": 1200.00}`

This contract ensures that any service needing product information knows exactly how to request it, and what data format to expect, without being exposed to the Product Service's internal complexities.

Knowledge Base: Inter-Service Communication

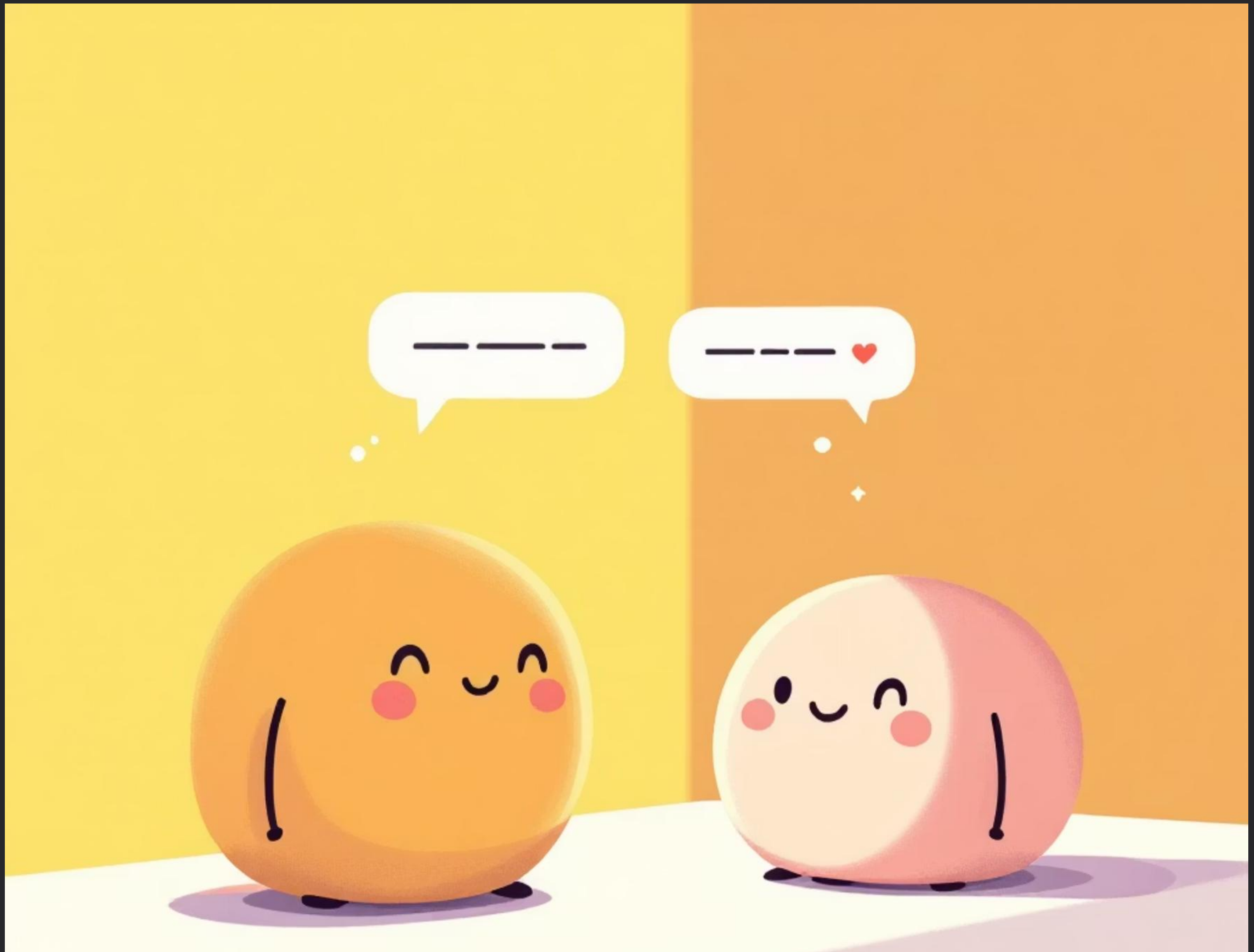
Microservices, by their nature, must communicate to fulfill requests and complete business processes. Understanding the different communication patterns is crucial for designing a resilient and decoupled system.

Type 1: Synchronous Communication (Request/Response)

Mechanism: Typically HTTP/REST, similar to how components in a monolith might communicate internally or how a client interacts with a server. The calling service makes a request and waits for an immediate response.

Use Case: Ideal for actions that require an immediate result, where the calling service cannot proceed without the response. For example, a Cart Service might synchronously call the Product Service to get the most up-to-date price before processing an item addition.

Drawback: Introduces **temporal coupling**. If the called service is down or slow, the calling service blocks and may eventually fail. This can lead to cascading failures across the system.



Type 2: Asynchronous Communication (Events/Messaging)

Mechanism: Involves message brokers (e.g., RabbitMQ, Kafka). A service publishes an event (a message) to a message broker without waiting for an immediate response. Other interested services consume these events at their own pace.

Use Case: Suited for actions that can be processed later or by multiple consumers without an immediate direct reply to the publisher. For instance, after an Order Service places an order, it publishes an "Order Placed" event. A Notification Service can consume this to send an email, and an Inventory Service can consume it to deduct stock.

Benefit: Promotes **high decoupling** and **fault tolerance**. The publisher isn't affected if consumers are temporarily unavailable. It also enables fan-out scenarios where multiple services react to a single event.

Activity 3: Designing Communication Strategy

To ensure our microservices architecture is resilient and efficient, we must strategically decide on the communication type for each inter-service interaction. Our primary goal here is to uphold the **Fault Isolation ASR (ASR 2)**, minimizing the impact of one service's failure on others.

Checkout Price Check	Cart Service → Product Service	Synchronous	The user's checkout experience critically depends on accurate and current pricing. The Cart Service **must** fail immediately if the price is unavailable, incorrect, or if the Product Service cannot be reached. This prevents fraudulent transactions or bad user experiences.
Order Confirmation	Order Service → Notification Service	Asynchronous	Email or SMS delivery is a secondary concern compared to the core business transaction (order placement). The Order Service should not be blocked if the Notification Service is temporarily unavailable. The message can be retried later, increasing overall system resilience.
Inventory Update	Order Service → Inventory Service	Asynchronous	Stock deduction can safely happen shortly **after** the order has been confirmed. Decoupling this process allows the Inventory Service to process updates at its own pace and handle potential backlogs without impacting the critical order placement flow. This enhances system resilience.

By carefully considering the criticality, immediacy, and fault tolerance requirements for each interaction, we can design a robust communication strategy that balances performance with resilience.