Development of Web Applications

Lecture 14

Today

- What’s the catch with monolithic systems?
- What is an architectural quantum?
- The fundamentals of microservice architecture
- Concepts: granularity, independence, boundaries
- Challenges: consistency, testing, orchestration
- Event sourcing and saga pattern
- Tools and technologies for microservices
- When (not) to use microservices

Problems with Monolithic Solutions

- A monolithic solution is written in a single programming language.

  What if we have two developers, each highly proficient in different languages?

- Today, there's often a need to scale parts of a system.

  Can a monolithic application scale a specific business function independently?

- A failure in one part of the monolith can crash the entire system.

  How can we prevent this reliably?

- Can we develop and deploy each business function independently?

Monolithic Architecture

- The application is developed and executed as a **single program component** – typically an executable file like a Windows .exe, or a DLL host
- It contains **all business functionalities**
- Advantages:
  - Simple deployment
  - Simple local development environment
- Disadvantages:
  - Maintenance becomes difficult as the app grows, or even impossible past a certain size
  - Even small changes require rebuilding and testing the **entire** application
  - Scalability is limited

Layered Monolith

- Organized into layers:
  - Presentation Business Layer -> Data Access Layer
  - UI -> Application -> Domain -> Infrastructure
- Each layer depends on the one below (e.g., presentation layer depends on the business layer)
- Helps maintainability through separation of concerns
- Still a **single component**, one build, one deployment
- Also referred to as a single **architectural quantum**

Architectural Quantum

- An independent component with high functional cohesion.
- The smallest architectural unit that can be independently developed, tested, and delivered.Additionally, an architectural quantum can be independently
  - Versioned
  - Compiled/built
  - Deployed
  - Scaled
- Examples:
  - Monolithic application (.exe) – one large quantum
  - Lambda function (e.g., AWS Lambda, Azure Function)
  - Docker container for a specific purpose (e.g., periodic batch job)

Introduction to Microservices Architecture

Microservices architecture is an approach to software development where a system is composed of a set of small, independent services, each implementing a specific business functionality.

Key concepts:

- **Independent components**: each (hopefully) with a clear responsibility
- **Distributed**: services communicate (e.g., via RESTful APIs)
- **Technological heterogeneity**: components can use different languages and platforms
- **Independent deployment and scaling**: each service is developed, tested, and deployed independently
- *Business functionality == Implemented business capability within a bounded context*

Quantum Characteristics of Microservices Architecture

- In microservices architecture, the goal is for each business function to be a **separate architectural quantum**
- Each service has its own lifecycle (build, test, deploy, scale)
- This provides high independence and system flexibility
- Examples:
  - **OrderService** as a microservice that manages orders with its own DB and REST API → a **quantum**
  - **InventoryService**, **UserService**, **NotificationService** – all separate quanta
  - *What is the smallest architectural unit? Some argue it's a single function like handleOrder(), other say it's a service like OrderService* ☺
- Debatable, depends on organization structure, team size, DevOps maturity, business boundaries (bounded contexts)

Characteristics of Microservice Architecture

- Business functions developed as separate services: **Orders**, **Inventory**, **Delivery**, **Payment**
- Services **communicate via messages** (synchronously or asynchronously)
- Each service is developed by its **own team** – sometimes even **a single person**
- Each service may use its **own tech stack**:

- *Languages: .NET Core, Java, Node.js, PHP, Python, Go, Rust...*
- *Technologies: Docker, Kubernetes, API Gateway, RabbitMQ, Kafka...*

Microservice Challenges

- Fault tolerance and data consistency: what if part of the system fails?
- Testability: how to test a distributed system locally?
- Management and orchestration: how to control 10+ services?
- Versioning and compatibility: what if a service changes its API?
- Deployment: how to release changes across multiple services in sync?

Failures and Consistency

What if a service "goes down"?

- ACID-style atomicity is no longer available as in monoliths, so transactions are not a real solution any more
- Example with 2 microservoces:
  - **OrderService** – stores an order in its own database
  - **InventoryService** – updates inventory stock
  - If OrderService succeeds but InventoryService fails (e.g., network error), the system ends in an **inconsistent state**.
- To solve this, we can use two new concepts:
  - **Event Sourcing** – events are permanently stored, and state is derived from event history
  - **Event-driven Saga Pattern** – distributed transactions broken into coordinated, yet independent steps with **compensation** logic

Failures and Consistency: Event Sourcing

Event Sourcing models all state changes as a sequence of events

- Instead of persisting state, the system records all events that led to it
- The current state is reconstructed by „replaying” events
    - Allows replay of sequence or a single events and reactive processing
- Transparency: complete audit trail of changes

Failures and Consistency: Saga Pattern

Saga Pattern manages distributed transactions through a sequence of local transactions

- Each local transaction emits an event triggering the next step
- If a step fails, a **compensating action** is triggered

Example:

- **OrderService** creates an order
- **InventoryService** reserves items
- **PaymentService** charges the customer
- If PaymentService fails → InventoryService **cancels the reservation**
- *Types: choreographed saga (event-based, each service reacts on its own) and orchestrated saga (one service controls the sequence)*

Testability

- For testing purposes, it's hard to spin up a local instance of the system due to many independent components; we need some kind of **isolation**
  - **Unit tests** with mocks/stubs
  - **Integration tests** need real service instances or tools like Docker Compose
- Even local dev may require orchestration tools (e.g., Kubernetes)
- Asynchronous communication (e.g., message queues) makes behavior harder to verify
- Possible solutions:
  - **Consumer-driven contracts** (e.g., Pact) – JSON defines consumer/provider expectations (e.g., GET /user/1)
  - **Service virtualization** or sandbox environments – simulate full endpoints (e.g., WireMock)
  - **In-memory emulation** – replace external DB/service/queue with fast in-memory versions (e.g., InMemoryDatabase)
  - **CI/CD pipelines** – automatically set up integration environments

Management and Orchestration

How do we manage systems with 10+ components?

Required tools:

- **Service registry** (e.g., Consul, Eureka)
- **API Gateway** (e.g., YARP, Kong, Ocelot)
- **Centralized logging** (e.g., ELK stack, Grafana Loki)
- **Health checks & monitoring** (e.g., Prometheus, Grafana)
- **Tracing** (e.g., OpenTelemetry, Jaeger)

Architectural Elements

- Communication:
  - REST (e.g., GET /orders/123)
  - gRPC (binary, highly performant)
  - Messaging (e.g., RabbitMQ)
- Database approaches:
  - One per service
  - Shared DB
- Authentication:
  - Token-based (e.g., JWT)
  - API Gateway enforces auth centrally
- Configuration:
  - Externalized (e.g., Azure App Config)

Discussion

- Is a microservice the smallest architectural unit?
- Is microservice architecture overkill for simple applications?
  - If you don't know exactly why you need microservices – don't implement them.
- Is it better to have 5 poorly maintained services, or 1 well-understood monolith?
- Would you rather need to update a massive monolith, or deal with a distributed system of 5 complex services, only one of which you need to know in details?

Conclusion

- Microservices are not a "silver bullet."
- The right architectural style choice depends on:
  - Team size
  - System complexity
  - Scalability needs
  - *Also: maturity of your architecture and DevOps*

The key is to understand **when** and **why** microservice architecture is better then a monolith!

Thank you for your attention!