Design Microservices Architecture with Patterns & Best Practices

Step-by-step process for software system design and evolve from monolithic to microservices following the patterns & principles

Refactor architectures with different aspects of microservices pillars

Mehmet Ozkaya

Architecture Design Journey

Architecture Design – Vertical Considerations

Way of Learning – The Course Flow

Design: First Version of Monolithic Architecture

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

3-Tier Architecture

Patterns &Principles

DRY

KISS

YAGNI

Persistent Storage

API Design Best Practices

Non-FR (Update)

Decoupled Presentation Layer

API-Driven Backend

Diverse Clients

Scalability of Tiers

Constraints

SPA development expertise

API design becomes critical

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop v2.0: A 3-Tier Architecture

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

3-Tier Architecture

Modular Monolith Architecture

Patterns &Principles

KISS, YAGNI, DRY, Separation of Concerns (SoC)

Persistent Storage

API Design Best Practices

Avoid «Big Ball of Mud»

Non-FR (Update)

High Internal Cohesion, Low Inter-Module Coupling

Team Autonomy

Maintainability & Testability

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop: MMA Architecture

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

HTTP-Based REST, gRPC, GraphQL

Gateway Routing, Aggregation, Offloading Pattern

API Gateway Pattern

Non-FR (Update)

High Scalability
High Availability

Handle Millions of Concurrent Users/Requests

Independent Deployability

Decoupled Clients from Backend

Centralized Cross-Cutting

Efficient Client Data Retrieval

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop Microservice with an API Gateway

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

API Gateway/ BFF Pattern

Event-Driven Architecture (EDA)

Publish/Subscribe Pattern (Fan-Out)

Non-FR (Update)

High Scalability
High Availability

Handle Millions of ConcurrentRequest

Independent Deployability

High Decoupling between Services
Resilience & Fault Tolerance

Scalability of Event Processing
Eventual Consistency

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ Event Bus for Asynchronous Communication

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes

AppHost + Projects and codes, put break points

Run App

See Aspire Dashboard and graph -projects env variables

Test

E2E use cases see products, add basket, checkout, see orders

Observe

Logs, Traces, Db records, caches, buses

Code: E-Shop Microservices Architecture

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

9-eshop-microservices-async-rabbitmq

DEMO: Microservices Architecture Code Review

DEMO: Code review of Microservices Architecture .NET Implementation

- https://github.com/mehmetozkaya/Design-Microservices-Architecture-with-Patterns-Principles

Why This Course?

90% Hands-on Design & **Code** Activities

Iterate Design Architecture from On-Premises to Cloud Serverless architectures

Evolves architecture monolithic to event-driven microservices and serverless

Refactoring System Design for handling millions of requests

Apply **best practices with microservices design patterns** and principles

Examine **microservices** patterns **with all aspects** like Communications, Data Management, Caching and Deployments

Prepare for **Software Architecture** & **System Design Interviews**

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

REST, gRPC, GraphQL

Gateway Routing, Aggregation, Offloading

API Gateway Pattern

Backends for Frontends (BFF)

Non-FR (Update)

High Scalability
High Availability

Handle Millions of Concurrent Users/Requests

Independent Deployability

Decoupled Clients from Backend

Optimized API for Each Client Type

Independent Evolution

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop Microservice with Dedicated BFFs

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture, EDA

Patterns &Principles

Database-per-Service

CAP Theorem, Database Sharding Pattern

Saga Pattern (Choreography/Orchestration)

Transactional Outbox Pattern, CDC

Cache Hit, Miss, Invalidation

Cache-Aside Pattern

Non-FR (Update)

High Scalability High Availability

Independent Deployability

Reduced Latency for Frequent Reads

Decreased Load on Primary Data Stores

Improved Application Responsiveness & Throughput

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ Central Distributed Cache

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

Shared DB Anti-pattern

Polyglot Persistence

CAP Theorem

Data Partitioning

Database Sharding Pattern

Non-FR (Update)

High Scalability High Availability

Handle Millions of ConcurrentRequest

Independent Deployability

Scalability for Specific DBs Stores High Availability & Fault Tolerance DBs

High Load for Data Operations

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ Database Sharding Pattern

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture, EDA

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

Polyglot Persistence

CAP Theorem

CQRS, Event Sourcing, Materialized View Pattern

Eventual Consistency

Non-FR (Update)

High Scalability High Availability

Handle Millions of ConcurrentRequest

Independent Deployability

High Write Throughput & Auditability

Optimized Read Performance

Scalability of Read and Write Paths

Resilience in Data

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ CQRS & Event Sourcing

Understanding Our Approach to Code in Design Course

This is a Software Architecture DESIGN Course

DESIGN

CODE

Review and analyze fully implemented code
Run, Debug and Test the application

DEMO: Microservices Architecture Code Review

DEMO: Code review of Microservices Architecture .NET Implementation

- https://github.com/mehmetozkaya/Design-Microservices-Architecture-with-Patterns-Principles

Why This Course?

90% Hands-on Design & **Code** Activities

Iterate Design Architecture from On-Premises to Cloud Serverless architectures

Evolves architecture monolithic to event-driven microservices and serverless

Refactoring System Design for handling millions of requests

Apply **best practices with microservices design patterns** and principles

Examine **microservices** patterns **with all aspects** like Communications, Data Management, Caching and Deployments

Prepare for **Software Architecture** & **System Design Interviews**

DEMO: Microservices Architecture Code Review

DEMO: Code review of Microservices Architecture .NET Implementation

- https://github.com/mehmetozkaya/Design-Microservices-Architecture-with-Patterns-Principles

The Shifting Landscape of Software Architecture

Architecture Design Journey

Our Evolutionary Path – From Monolith to Serverless

Iterate Design: From **On-Premises** thinking to **Cloud-Native & Serverless**

Evolve Architectures: Monolithic to highly **Scalable & Event-Driven**

Refactor for Scale: Designing to handle **millions of requests**

Apply Best Practices: Using established **design patterns & principles**

Macroservices to Nanoservices

Which Architecture Approach We Should Choose ?

It depends on your project

Use Mixed Architecture

Provide transitions between architectures

Architecture Design – Vertical Considerations

Way of Learning – The Course Flow

Way of Learning – The Course Flow

- Identify Clearly

- Architecture Approaches
- Patterns
- Principles & Best Practices

- Fill Design Toolbox
- Design Initial version

- Develop Eshop App
- Explore Codebase
- Run the App

- What's wrong with this architecture ?
- How we can improve this current design ?

Our Starting Point

Problem: We need to sell products online simply

Learn: E-commerce basics, What are the NFRs? What's a Monolithic architecture?

Design: Design a basic Monolithic E-Shop

Code: .NET code for this Monolith. How is it structured?

Evaluate: Pros: Simple to start. Cons: What happens when traffic grows? What if different parts need to scale independently? -> Leads to a NEW PROBLEM.

Problems We'll Tackle

Monolith Can't Scale: Handling Increased Traffic

Need for Independent Deployments: Breaking Down the Application

Client Complexity: Simplifying API Access

Chatty Services: Reducing Network Load

Data Silos: Querying Across Services

Slow Operations: Beyond Synchronous Calls

Keeping Data Consistent in Distributed Systems

Zero-Downtime Deployments & Flexible Scaling

Problem: Challenges of a Tightly Coupled UI & Backend

Development Bottlenecks

Technology Lock-in

Scalability Issues

Omnichannel Difficulty

Slower UI Innovation

Team Specialization

Challenges

Solution:

3-Tier Architecture

Problem: Monolith is Becoming Hard to Manage and Evolve

The Underlying Need: Better Internal Structure and Modularity

Need a way to organize our monolithic codebase into well-defined, loosely coupled internal modules

Solution:

Modular Monolithic Architecture

Problem: Monolith Bottlenecks at Scale

Problem: Monolith Bottlenecks - Slowing Innovation, Straining Teams, and Limiting Independent Scaling

A single, large codebase and deployment unit, even if scaled, becomes a chokepoint for rapid, independent evolution and fine-grained scaling

Problem: How to Decompose Microservices Effectively?

Are these services truly independent?

Do they encapsulate a coherent set of responsibilities?

Do their boundaries align with our business domain and team structures?

Are they an appropriate size?

Problem: How to Communicate Microservices Effectively?

What communication protocols will they use (HTTP, gRPC, AMQP)?

What message formats will they exchange (JSON, XML, Protocol Buffers)?

How will services find each other on the network?

How will they handle API versioning as services evolve independently?

How do we design these service-to-service APIs to be efficient and avoid chattiness?

Problem: Direct Client-to-Service Communication

Problems

- Increased Client-Side Complexity
- Cause to chatty calls from client to service
- Hard to manage invocations from client app with all different protocols (HTTP, GraphQL, gRPC)
- Duplication of Effort Across Multiple Client Types
- Security & Cross-Cutting Concerns

Solutions

- Well-defined API Design
- **Microservices Communication Patterns Patterns (API Gateway, BFF, Publish/Subscribe..)**

Problem: Client Apps has Different UI Requirements

Problem: Tailoring Data for Diverse Clients - Generic APIs Create Client-Side Burdens

A one-size-fits-all API from backend microservices often forces client applications to do extra work or fetch unnecessary data

Problem: Long Running Operations Can't Handle with Sync Communication

Problems

Increased Latency & Poor Performance

Reduced Availability & Lack of Fault Tolerance

Tight Coupling Between Services

Resource Hogging

Best Practices

True Decoupling, Minimize communication between the internal microservices

Enhanced Resilience

Non-Blocking Operations

Solution

Asynchronous Message-Based Communications

Working with events

Problem: Databases Create Bottlenecks When Scaling

Problems

Databases are inherently stateful

Traditional Vertical Scaling Has Limits

Provide Data Consistency Across Multiple Databases is Hard

Read/Write Bottlenecks, Connection Limits

Solution

Data management strategies

Handle high read AND write loads

Scale horizontally more easily

Exploring different Database Types (SQL vs. NoSQL), Data Partitioning/Sharding and Advanced Replication/Clustering

Problem: Cross-Service Queries and Write Commands on Distributed Scaled Databases

Considerations

Cross-services queries that retrieve data from several microservices ?

Separate read and write operations at scale ?

Problems

Cross-Service Queries with Complex JOIN operations

Read and write operations at scale

Distributed Transaction Management

Solutions

Microservices Data Query Pattern and Best Practices

Materialized View Pattern

CQRS Design Pattern

Event Sourcing Pattern

Problem: Manage Distributed Transactions and Consistency between Microservices

Considerations

Distributed Transactions that required to visit several microservices ?

Consistency across multiple microservices ?

Rollback transaction and run compensating steps ?

Problems

Distributed Transaction Management

Rollback Transaction on Distributed Environment

Run Compensate Steps if one of service fail

Solutions

Saga Pattern for Distributed Transactions

Transactional Outbox Pattern

Compensating Transaction pattern

CDC - Change Data Capture

Problem: The Cost of Frequent Data Store Access

Problems

Database Operations Can Be Expensive

Repetitive Access to the Same Data

Latency in Event-Driven Flows

Performance of Synchronous API Calls

Considerations

Reduce latency for common read operations

Decrease load on primary data stores

Improve overall system responsiveness and throughput

Solutions

Distributed Caching

Storing frequently accessed data in a distributed cache

Problem: Deploy Microservices at Anytime w/ Zero-downtime and flexible scale

Problems

Manually deploying, updating and scaling a large number of services is error-prone, time-consuming

Considerations

Need for Zero-Downtime Deployments

Demand for Agile, Flexible Scaling

Complexity of Managing Many Moving Parts

Ensuring Consistent Environments

Solutions

Containers and Orchestrators

Deployment strategies; blue-green, rolling and canary deployment

Kubernetes Patterns; Sidecar Patterns, Service Mesh

DevOps and CI/CD Pipelines and IaC

DEMO: Microservices Architecture Code Review

DEMO: Code review of Microservices Architecture .NET Implementation

- https://github.com/mehmetozkaya/Design-Microservices-Architecture-with-Patterns-Principles

Understanding Our Approach to Code in Design Course

This is a Software Architecture DESIGN Course

DESIGN

CODE

Review and analyze fully implemented code
Run, Debug and Test the application

Prerequisites – What You Should Know

[.NET 9 (Includes .NET Aspire by default)]

[Visual Studio 2022 or Later]

[Docker Desktop]

Slide Source Code on GitHub

All the code written and demonstrated in this course is available on GitHub

- https://github.com/mehmetozkaya/Design-Microservices-Architecture-with-Patterns-Principles

Review the source codes section-by-section

Course Slides and All Resources

Download Course Slides and **All Resources** from **This Lecture Resources**

Your Feedback is Invaluable! (Please Review!)

Your Review Powers This Course & Future Content!

Please take a moment to leave a 5-star review and a comment

Way of Learning – The Course Flow

Way of Learning – The Course Flow

First Problem: Sell Products Online

Create E-Commerce Web Application

Browse, select and purchase products

Avaliable 7/24

Handle good amount of request per second

Provide acceptable latency for users

Understand Problem

Understand E-Commerce Domain

Functional & non-functional requirements

Use cases

Understand EShop Domain: FRs, Use Cases

Goal: Create an Eshop Web Application

Users can view a list of products

Users can view product details

Users can add products to a shopping basket

Users can place an order for the products in their bask

Understand EShop Domain: Non-Functional Requirements

-ilities

Available to users **24/7**

"good amount" of **requests per second**

Acceptable response time (latency)

Orders placed should be **processed accurately**

Way of Learning – The Course Flow

Monolithic Architecture

Benefits and Challenges of Monolithic Architecture

Design our E-Shop application with Monolithic Architecture

Mehmet Ozkaya

First Problem: Sell Products Online

Create E-Commerce Web Application

Browse, select and purchase products

Avaliable 7/24

Handle good amount of request per second

Provide acceptable latency for users

Understand Problem

Understand E-Commerce Domain

Functional & non-functional requirements

Use cases

Architecture Design Journey

What is Monolithic Architecture ?

A single, unified codebase and deployment unit containing all application functionalities

What is Monolithic Architecture ?

Traditional approach to software development

Developing a complete application as a **single unit**

UI, Business and DB calls is in **single**, **same codebase**

All application ships with **single big deployment** with single jar/war file

Most of legacy apps are mainly implemented as a monolithic architecture

Can’t say old style arcitecture, **still valid** for particular scenarios

When is Monolithic Architecture a Good Choice ?

Small, Focused Teams & Startups
Rapid development, less coordination overhead

Simple Applications with Predictable Scale
When complexity and scalability needs are modest

Proof of Concepts (PoCs) & Quick Launches
Validating ideas fast, iterating quickly

Limited Microservices Expertise
Leverage existing team skills

Strong Consistency Requirements
Simpler to achieve in a single unit

Our Initial E-Shop Application

Initially, expected 2,000 concurrent users and 500 requests per second

The Benefits & Strengths of Monolithic Architecture

Simpler to Develop
Standard approach, readily available skills

Easier Debugging & Testing
Single codebase, direct calls, faster end-to-end tests

Straightforward to Deploy
Single artifact, less complex deployment process

Simplified Initial Scaling (Vertical)
"Scaling up" by adding more resources to the server

Lower Initial Operational Overhead
Fewer moving parts to manage

Performance
No network latency for internal calls

The Challenges & Limitations of Monolithic Architecture

Growing Complexity Over Time
Can become a "big ball of mud," hard to understand

Slower Development Speed
Changes become riskier, harder to implement, fear of breaking things

Deployment Bottlenecks
Small change requires redeploying the entire application

Barrier to New Technology Adoption
Difficult to introduce new frameworks/languages

Scalability Limitations
Cannot scale individual components independently

Reliability Issues
A bug in one module can bring down the entire app

What's wrong with traditional monolitic architecture?
Lose control monolitiic architecture
Lose control to manage , scale, and update
Hard to manage to manage, and update.

Characteristics of Monolithic Architecture

Single Codebase

- Easier to manage and deploy initially.
- No separation between different parts of the app.

Single Deployment Unit

- Entire app is packaged and deployed as a single unit.
- Simplify deployment processes, especially in the early stages of development.

Tight Coupling

- Changes in one part of the application can affect other
- Increased complexity and difficulty in making changes scaling specific parts of the application.

Shared Database

- Uses a single, shared database for all its data needs.

Big Ball of Mud

- Software system with a **messy**, **poorly structured** architecture. The code is so entangled that it's hard to understand, maintain, or extend.
- **No clear structure** or **organization**. Everything is mixed together, making it hard to figure out where one part ends and another begins.
- Over time**, quick fixes** and **shortcuts add up**, creating **technical debt**. This makes the code messy and difficult to work with.
- In a **rush to deliver features quickly**, best practices are often ignored. This leads to a **messy codebase** full of shortcuts.

Summary: The Monolith - A Foundational Tool

A valuable starting point, understand its trade-offs for informed future evolution.

The Architect's Challenge: Managing Complexity

It's easy to over-engineer. These principles help us build smarter, not just harder.

KISS - "Keep It Simple, Stupid"

KISS: Keep It Simple, Stupid

Simplicity should be a key goal

Avoid unnecessary complexity in design and implementation

Simple systems are **easier** to **understand**, **maintain**, and **debug**

YAGNI - "You Ain't Gonna Need It"

YAGNI: You Ain't Gonna Need It

Don't implement functionality based on **future speculation**

From the **Extreme Programming (XP)** methodology: the **simplest thing that could possibly work**

Focus on **current**, **concrete requirements**

Avoids wasted effort on **features** that may **never be used**

DRY - "Don't Repeat Yourself"

DRY: Don't Repeat Yourself

Every piece of knowledge must have a **single**, **unambiguous**, **authoritative representation** within a system

Avoid duplicated code, **configuration** or **design** elements

Promotes **maintainability**, **reduces errors** and **minimizes complexity**

Applying These Principles to Our E-Shop Design

KISS: We'll aim for the simplest design that meets the E-Shop's current needs

YAGNI: We won't build features for hypothetical future E-Shop scenarios

DRY: We'll look for opportunities to avoid redundancy in our E-Shop's modules and components

Way of Learning – The Course Flow

The Architect's First Step: Opening the Design Toolbox

Functional Requirements (FRs)

Non-Functional Requirements (NFRs)

Constraints (Time, Cost, Team)

Learned Architectures

Patterns & Principles

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Patterns &Principles

DRY

KISS

YAGNI

Non-FR

Availability

Small number of Concurrent User

Acceptable response times

Simplicity

Constraints

Time: Quick

Team: Small

Budget: Limited

Assumption: Idea

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design: First Version of Monolithic Architecture

Why This Simple Design is "Good Design" (For Now)

Good architecture is about fitness for purpose, not complexity.

E-Shop's Development Technology Stack

Language: C#

Framework: .NET / ASP.NET Core (for web application)

IDE/Tooling: Visual Studio

Alternatives: Java Spring Boot, Python Django, Node.js, Go, etc.

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: EShop Monolithic App w/ KISS & YAGNI

Language: C#

Framework: .NET / ASP.NET Core (for web application)

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

1-eshop-monolith-one-app

Problem: Customer Basket Items Disappeared When App Restarts

Add some items to the shopping basket, and then simulate an app restart

Crash, a new deployment, or routine maintenance

Solution

A Database

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Patterns &Principles

DRY

KISS

YAGNI

Persistent Storage

Non-FR

Availability

Small number of Concurrent User

Acceptable response times

Data Persistence & Durability

Constraints

Time: Quick

Team: Small

Budget: Limited

Assumption: Idea

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design: First Version of Monolithic Architecture

E-Shop's Development Technology Stack

Language: C#

Framework: .NET / ASP.NET Core (for web application)

IDE/Tooling: Visual Studio

Alternatives: Java Spring Boot, Python Django, Node.js, Go, etc.

Relational Databases: PostgreSQL, MySQL, SQL Server, Oracle

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Monolithic App with PostgreSQL Database

Language: C#

Framework: .NET / ASP.NET Core (for web application)

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

2-eshop-monolith-with-db

Way of Learning – The Course Flow

Our Current Monolith: UI and Backend in One Unit

User Interface, Business Logic, and **Data Access** are all part of **the same application** and **deployment unit**

Problem: Tightly Coupled UI Limits Flexibility and User Experience

When UI and backend logic are one, evolving them independently becomes difficult

Web Browser UI

Mobile App Needs API?

Partner System Needs API?

Problem: Challenges of a Tightly Coupled UI & Backend

Development Bottlenecks

Technology Lock-in

Scalability Issues

Omnichannel Difficulty

Slower UI Innovation

Team Specialization

Challenges

Solution:

3-Tier Architecture

Way of Learning – The Course Flow

What is 3-Tier Architecture?

Why Use 3-Tier Architecture?

Clear Separation of Concerns
Each tier has a distinct role

Independent Scalability
Scale each tier based on its specific load

Technology Flexibility per Tier
Use the best tech for each job

Maintainability & Development
Teams can focus on specific tiers; changes are more isolated

Security
Place application tier behind firewalls, protecting the data tier

Reusability of Business Logic
APIs from the Application Tier can serve multiple Presentation Tiers

How 3-Tier Solves Our UI Coupling Problem for E-Shop

Presentation Tier (SPA) becomes a **separate application**

Application Tier (our monolith) now primarily **exposes Web APIs**

Enables choice of **modern SPA frameworks** (**React, Angular, Vue**) for rich UIs

Backend API can serve **multiple clients** (Web SPA, future mobile apps)

Layers vs. Tiers

N-Layer

Logical code organization

3-Tier

Physical/Deployment separation of those logical concerns

External vs. Internal Architecture

Summary of 3-Tier Architecture

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

3-Tier Architecture

Patterns &Principles

DRY

KISS

YAGNI

Persistent Storage

SoC

API Design Best Practices

Non-FR (Update)

Decoupled Presentation Layer

API-Driven Backend

Diverse Clients

Scalability of Tiers

Constraints

SPA development expertise

API design becomes critical

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop v2.0: A 3-Tier Architecture

E-Shop's Development Technology Stack

Presentation Tier (SPA): Blazor, React, Angular, Vue.js

Application Tier (Backend API): ASP.NET Core Web API, .NET

Data Tier (Database): PostgreSQL

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Monolithic App with PostgreSQL Database

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

3-eshop-monolith-add-ui

Big Ball of Mud

- Software system with a **messy**, **poorly structured** architecture. The code is so entangled that it's hard to understand, maintain, or extend.
- **No clear structure** or **organization**. Everything is mixed together, making it hard to figure out where one part ends and another begins.
- Over time**, quick fixes** and **shortcuts add up**, creating **technical debt**. This makes the code messy and difficult to work with.
- In a **rush to deliver features quickly**, best practices are often ignored. This leads to a **messy codebase** full of shortcuts.

Way of Learning – The Course Flow

Our Current Backend API: Still a Monolith Inside

Application Tier (Backend API) successfully **serves** our **SPA**, but **internally**, its codebase is still **one large, undifferentiated unit**

Problem: Monolith is Becoming Hard to Manage and Evolve

The Business is **Booming!**

New Features, **Team** and **Codebase** Size **increasing**

As **business** and **teams scale**, an undifferentiated **monolith struggles** to keep up

Problem: Unstructured Growth - Our Monolith is Becoming Hard to Manage and Evolve

Challenges of a Growing Monolithic Backend

Code Becomes Too Complex Over Time
"Big Ball of Mud" scenario

Hard to Understand Codebase
Onboarding new developers becomes painful

Lack of Code Organization
Changes in one area break another area

Reduced Development Velocity
Same large codebase can lead to merge conflicts, slower builds

Difficulty in Parallel Work
Teams can't easily work on their features independently

Increased Risk with Changes
Fear of making changes

Problem: Monolith is Becoming Hard to Manage and Evolve

The Underlying Need: Better Internal Structure and Modularity

Need a way to **organize** our **monolithic codebase** into **well-defined, loosely coupled internal modules**

Solution:

Modular Monolithic Architecture

Way of Learning – The Course Flow

Modular Monolith Architecture

Benefits and Challenges of Modular Monolith Architecture

Design our E-Shop application with Modular Monolith Architecture

Mehmet Ozkaya

Architecture Design Journey

Problem: Monolith is Becoming Hard to Manage and Evolve

The Business is **Booming!**

New Features, **Team** and **Codebase** Size **increasing**

As **business** and **teams scale**, an undifferentiated **monolith struggles** to keep up

Problem: Unstructured Growth - Our Monolith is Becoming Hard to Manage and Evolve

Modular Monolith (Modulith) Architecture

Developed as a **single**, **cohesive unit** but is **internally divided** into **well-defined**, **loosely coupled** modules

Breaks up the code into **independent modules** and each **module encapsulates** their **own features** needed in app

Still build and **deploy** a **single app**, breaking up the code into **independent modules**

Modules are represents **Bounded Context** of our app domain and **group features** of **Domain contexts**

Reduces the **dependencies** of a **module** and we can **develop** or **modify** a **module** without it **effecting other modules**

What is Modular Monolith (Modulith) Architecture ?

Designed and developed as a **collection** of **independent**, **interchangeable** and **loosely coupled modules**

Each module has a **specific business domain/capability**

Clear interfaces between **modules**, **strong encapsulation** within **modules**

Still **deployed** as a **single unit**

When and Why Should We Use a Modular Monolith?

When **Existing Monolith** is **Becoming Too Complex**

To **Improve Team Autonomy** and **Parallel Development**

When **Need Clear Boundaries** but want to **Avoid Distributed System Complexities**

As a **Potential Stepping Stone** Towards **Microservices**

Benefits of Modular Monoliths

Code Organization and Understandability

Maintainability, Testability

Team Autonomy

Simpler Deployment (Compared to Microservices)

No Distributed System Overhead

Facilitates Future Evolution

EShop with Modular Monolithic Architecture

Example Use Cases

Listing products = Product Module

Add to basket = Basket Module

Checkout order = Order Module

Main Benefits

Reduced complexity

Easier to refactor

Better for teams

Challenges of Modular Monoliths

Requires Strong Architectural Discipline

Risk of 'Leaky' Abstractions or Tight Coupling

Shared Resources

Scalability is Still Monolithic

Build Times Can Still Be Long

Technology Stack Homogeneity

EShop Modular Monoliths
API
Modules
Catalog
Public API
Basket
Public API
Public API
Ordering
Public API
Identity
PostgreSQL
Class Library
Modules
Catalog
Basket
Ordering
Shared
Shared
.NET 8
Module A
In-process Method Calls
Module B - Public API
Public API
Module B Contracts
Module B

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

3-Tier Architecture

Modular Monolith Architecture

Patterns &Principles

KISS, YAGNI, DRY, Separation of Concerns (SoC)

Persistent Storage

API Design Best Practices

Avoid «Big Ball of Mud»

Non-FR (Update)

High Internal Cohesion, Low Inter-Module Coupling

Team Autonomy

Maintainability & Testability

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop: MMA Architecture

E-Shop's Development Technology Stack

Presentation Tier (SPA): Blazor, React, Angular, Vue.js

Application Tier (Backend API): .NET / ASP.NET Core We

Data Tier (Database): PostgreSQL

Why Is DB Schema Separation Important ?

In a **modular monolith architecture**, **maintaining clear boundaries** between **different modules** is **essential**.

Database schema separation is a **practice** that helps achieve this **by isolating the data** of each **module** within **its own schema**.

Best Practice: Separate Schema

For our **EShop Modular Monoliths** application, we will follow the "**Separate Schema**" approach.

Logical Data Isolation

Each module's data is stored in a separate schema, making it clear which data belongs to which module.

Maintain the integrity and consistency of the data.

Provides a good balance between isolation and ease of management.

Ensures logical separation of data, which simplifies maintenance and security.

Best Practice: Separate Schema

Ease of Management

Easier to manage compared to multiple databases or different persistence mechanisms.

Schema separation reduces the complexity of managing the database.

Changes in one module's schema do not impact other modules, making it easier to evolve and maintain the application.

Simplifies backup, restore, and migration processes.

Best Practice: Separate Schema

Performance

Reduces the complexity of cross-module queries.

Shared database resources can be optimized more effectively.

Security

Allows for fine-grained access control at the schema level.

Simplifies auditing and compliance requirements.

Implement Separate Schema in EShop Modular Monoliths

Multi-context EF DB Context for DB Separate Schema

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Monolithic App with PostgreSQL Database

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

4-eshop-modular-monoliths-first

EShop Modular Monoliths
API
Modules
Catalog
Public API
Basket
Public API
Public API
Ordering
Public API
Identity
PostgreSQL
Class Library
Modules
Catalog
Basket
Ordering
Shared
Shared
.NET 8
Module A
In-process Method Calls
Module B - Public API
Public API
Module B Contracts
Module B

Monolithic Architecture Vertical Considerations

Communication Within a Monolithic Architecture

Primarily: **In-Process Communication**

Modules **reside** in the **same application process**

Communication is typically via **direct method calls**

Benefits: Fast, Simple, No Network Overhead

Asynchronous queues or even **external message brokers** for specific **async tasks** within the **monolith**

EShop Modular Monoliths
API
Modules
Catalog
Public API
Basket
Public API
Public API
Public API
Ordering
Identity
PostgreSQL
EShop Modular Monoliths
API
Modules
Catalog
Public API
Basket
Public API
Public API
Identity
PostgreSQL
Message Broker
Separated Service
Ordering
Module A
In-process
Method Calls
Module B - Public API
Public
API
Module B
Contracts
Module B

Monolithic Architecture Vertical Considerations

Transaction Management in Monolithic Architecture

Generally simpler due to a **single database** and **process**

Leverages **ACID properties** of relational databases

Frameworks/ORMs provide **built-in transaction** mechanisms (e.g. Unit of Work)

Easy to ensure **atomicity** for operations **spanning multiple internal components**

```
function place_order()
    do_payment
    decrease_stock
    send_shipment
    generate_bill
    update_order
```

Transaction Management in Monolithic Architecture - 2

Operated in the **transaction scope** are kept in memory without writing to the database until they are **committed**

Ifa **Rollback** is made at any time, all transactions in the scope are **deleted** from memory and the **transaction** is **canceled**

```
function place_order()
    do_payment
    decrease_stock
    send_shipment
    generate_bill
    update_order
```

Monolithic Architecture Vertical Considerations

Deployments of Monolithic Architecture

Single deployment unit for the **entire app**

Deploying one thing can be easier than many

"**All or Nothing**" Deployments: Small change requires redeploying the entire application

A bug in one part can **affect** the **whole app**

Require downtime or more complex strategies (blue-green) for zero-downtime

The **entire monolith** is **scaled**, **not individual** features

Deployments of Monolithic Architecture

Summary: Monolith's Operational Trade-offs

Communication:

- (+) Fast, Simple (In-Process)
- (-) Not applicable for external system integration without APIs

Transactions:

- (+) Simple, ACID (Single DB)
- (-) Can become complex if business logic is tangled

Deployments:

- (+) Simpler for small apps (One unit)
- (-) Risky, Slow, All-or-Nothing (As app grows)

Way of Learning – The Course Flow

E-Shop is Getting Popular!

Success brings **new challenges**! Our E-Shop is **attracting more users**, especially **during sales** or **promotions**

What happens when **many users access** the **Product Catalog simultaneously**?

Problem: Product Page Slowness - Failures During Peak Traffic

Peak Time Blues - Product Page Slowness & Overload

Frequent, **repetitive database reads** for product data **become a bottleneck under high load**

Symptoms & Impact of Product Page Slowness

Slow page load times for product listings

App unresponsiveness during peak hours

Increased database CPU and I/O load

Timeouts or errors under heavy load

Poor User Experience & Frustration

Lost Sales & Revenue

Damage to Brand Reputation

Increased Infrastructure Costs

Goal: Reducing Repetitive Database Load for Reads

Serve **frequently accessed**, **relatively static data** (product page data) **without hitting the database every single time**

Solution: Common performance optimization strategy: Caching!

Way of Learning – The Course Flow

What is Output Caching?

Storing the **fully generated output** of a **request** (HTML page, API JSON response) in a **fast-access cache**

Subsequent **identical requests** can be **served directly** from the **cache**, **bypassing expensive processing**

How Output Caching Solves Our Product Page Problem

Key Considerations for Implementing Output Caching

Cache Duration (Expiration/TTL)
How long should items stay in the cache?

Cache Invalidation Strategy
How to update/remove cached items when underlying data changes

Cache Key Design
How to uniquely identify cacheable items

Cache Storage Location
In-memory (per server), Distributed Cache

What to Cache
Focus on frequently accessed, expensive-to-generate, relatively static content

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

3-Tier Architecture

Modular Monolith Architecture

Patterns &Principles

KISS, YAGNI, DRY, Separation of Concerns (SoC)

Persistent Storage

API Design Best Practices

Avoid «Big Ball of Mud»

Output Caching

Non-FR (Update)

High Performance & Low Latency

Reduced Database Load

Scalability for Read-Heavy Workloads

Consideration

Cache Invalidation Strategies

Cache Key Design

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop: Output Caching

E-Shop's Development Technology Stack

Cache Store: Redis, Memcached, Ncache

Presentation Tier (SPA): Blazor, React, Angular, Vue.js

Application Tier (Backend API): .NET / ASP.NET Core Web API

Data Tier (Database): PostgreSQL

ASP.NET

PostgreSQL

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Monolithic App with PostgreSQL Database

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

5-eshop-modular-monoliths-add-cache

Way of Learning – The Course Flow

Our E-Shop is a Hit - Traffic is Exploding!

Massive **increases** in **concurrent users**

Sustained **high request rates**, not just for product pages but **across the app**

Our current **single instance** of the **ApiService** is **becoming** a **bottleneck**

Problem: Single Instance Bottleneck - Can't Handle More Request

Problem: Traffic Surges Are **Overwhelming** Our **Single Application Instance**

A **single backend** server instance has **finite capacity** for **handling requests**, **CPU**, **memory**, and **I/O**

Symptoms & Impact of Overloaded Single Instance

Increased response times (latency)

High CPU and memory usage

Requests queueing up, leading to timeouts

ApiService instance to crash under extreme load

Poor User Experience & Frustration

Significant Lost Sales & Revenue

Damage to Brand Reputation

Inability to capitalize on peak demand

Goal: Increasing Application Tier Capacity

Need to **increase** the **processing capacity** of our **ApiService** to handle many **more concurrent users** and **requests**

Solution: This points towards strategies for **Application Scalability**

Way of Learning – The Course Flow

Learn: Scalability - Why need to Scale ?

E-Shop's single ApiService instance is getting overwhelmed by surging user traffic

Fundamental concept of Scalability

Vertical Scaling (Scaling Up) and Horizontal Scaling (Scaling Out)

Scalability - Vertical and Horizontal Scaling

Vertical Scaling - Horizantal Scaling of Application

How many concurrent request can accommodate our design ?

Load Balancer with Consistent Hashing

Mehmet Ozkaya

Problem: Single Instance Bottleneck - Can't Handle More Request

Problem: Traffic Surges Are **Overwhelming** Our **Single Application Instance**

A **single backend** server instance has **finite capacity** for **handling requests**, **CPU**, **memory**, and **I/O**

Non-Functional Requirements

-ilities

Scalability	Availability	Reliability
Maintability	Usability	Eficiency

What is Scalability?

The **ability** of a system to **handle** an **increasing amount of work** (**requests**, **users**, **data**)

Measured by the **number** of **requests** an application can **successfully handle simultaneously**

When a **system can't handle more**, it has reached its **scalability limit**

Two Primary Scaling Strategies: Vertical vs. Horizontal

Vertical Scaling (Scale Up): Add more power (CPU, RAM, DISK) to an existing machine

Horizontal Scaling (Scale Out): Add more machines to your pool of resources

Vertical Scaling (Scale Up)

Making a single node/server stronger

Upgrade existing hardware (CPU, RAM, Disk I/O)

Pros

Simpler to implement initially

App logic doesn't necessarily need to change

Data consistency is easier

Cons

Hardware Limits & Cost

Single Point of Failure

Downtime for Upgrades

Reach Scalability Limits

Horizontal Scaling (Scale Out)

Distributing the load across multiple machines/nodes

Pros

Higher Overall Scalability

Fault Tolerance & High Availability

Cost-Effective

Cons

Architectural Complexity

Requires a Load Balancer

Managing state can be challenging

Inter-instance communication adds network latency

Summary: Choosing the Right Scaling Strategy

Vertical Scaling (Up): Good for moderate increases, simplicity (initially), when application isn't easily distributable. Limited by hardware/cost

Horizontal Scaling (Out): Best for high scalability, fault tolerance, cloud environments. Requires app design for distribution and a load balancer

What is a Load Balancer? The Traffic Manager

A **dedicated component** (hardware or software) that **acts** as a **'traffic cop'** for **your servers**

Sits between **clients** and a **pool** of **backend servers**

Distributes incoming network and **application** traffic **across multiple servers**

Goal: Improve responsiveness, availability, and prevent any single server from being overwhelmed

Key Functions and Benefits of Using a Load Balancer

Distributes Load
Prevents server overload, improves performance

Enhances Scalability
Enables effective horizontal scaling by adding/removing servers seamlessly

Availability & Fault Tolerance
Automatically reroutes traffic if a server instance fai

Session Persistence
Can direct a user's subsequent requests to the same server

SSL Termination
Can handle HTTPS decryption, offloading servers

Load Balancing Algorithms - How is Traffic Distributed?

Round Robin
Distributes requests sequentially to each server in the pool. Simple and common

Least Connections
Sends new requests to the server currently handling the fewest active connections

IP Hash
Uses the client's IP address to determine which server receives the request

Weighted Algorithms
Assigns weights to servers based on their capacity

Consistent Hashing - Smart Distribution for Dynamic Systems

Advanced algorithm ideal for **dynamic environments** where servers are **added/removed**

Minimizes data/request remapping when the server pool changes

Maps servers and **data keys** to a **conceptual ring**, **Data/request** goes to the **'next' server** on the **ring**

If a **server** is **added/removed**, only **keys/requests** for that **segment** of the **ring** are **affected**

Why Consistent Hashing is Powerful

Solves **Horizontal Scalability Problem Effectively**

Drastically **reduces cache misses** when cache servers are scaled

Better Load Distribution

Best option when **working** with **distributed systems**

Horizontal Scalability

(Scaling Out)

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

3-Tier Architecture

Modular Monolith Architecture

Patterns &Principles

KISS, YAGNI, DRY, Separation of Concerns (SoC)

Persistent Storage

API Design Best Practices

Output Caching

Horizontal Scaling (Scale Out)

Load Balancer

Non-FR (Update)

High Scalability for Read-Heavy Workloads

High Availability

Considerations

ApiService should be stateless

Load balancer configuration

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop: 3-Tier with a Horizontally Scaled Backend

E-Shop's Development Technology Stack

Presentation Tier (SPA): Blazor, React, Angular, Vue.js

Load Balancer: NGINX, HAProxy, Cloud LB (AWS ELB, Azure Load Balancer, Google Cloud LB)

Application Tier (Backend API): .NET / ASP.NET Core Web API

Data Tier (Database): PostgreSQL

Design - E-Shop: 3-Tier with a Horizontally Scaled Backend

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Monolithic App with Horizontal Scaling

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

5-eshop-modular-monoliths-add-cache

Way of Learning – The Course Flow

E-Shop Dominates, But at What Cost?

Our **E-Shop** is a **market leader**, but **success reveals** new **strains**

Business demands **RAPID feature releases** to stay ahead

Specialized development teams need to **work** in **parallel without friction**

Some parts of the app experience **vastly different load patterns** than others

Problem: Monolith Bottlenecks at Scale

Problem: Monolith Bottlenecks - Slowing Innovation, Straining Teams, and Limiting Independent Scaling

A single, large codebase and **deployment unit,** even if scaled, becomes a **chokepoint** for rapid, **independent evolution** and **fine-grained scaling**

Deployments of Monolithic Architecture

Specific Challenges of the Scaled Monolith Backend

Reduced Feature Velocity & Slower Time-to-Market

Decreased Team Autonomy & Increased Coordination Overhead

Inflexible, Coarse-Grained Scaling

Technology Stack Rigidity

Fault Isolation Issues

Can't Scale and Deploy Independently
Can't add new features immediately
Can't Deploy features immediately, waiting for deployment dates

Goal: True Independence for Teams, Features & Scaling

Allow **teams** to **develop**, **deploy** and **scale** their **specific business capabilities independently**

Ship new features faster **without risking the entire application**

Optimize resource usage by **scaling individual parts** of the system **based on their unique demand**

Solution: Microservices Architecture !!

Way of Learning – The Course Flow

Microservices Architecture

Benefits and Challenges of Microservices Architecture

When to use Microservices Architecture

Design our E-Commerce application with Microservices Architecture

Mehmet Ozkaya

Architecture Design Journey

Problem: Monolith Bottlenecks at Scale

Problem: Monolith Bottlenecks - Slowing Innovation, Straining Teams, and Limiting Independent Scaling

A single, large codebase and **deployment unit,** even if scaled, becomes a **chokepoint** for rapid, **independent evolution** and **fine-grained scaling**

Goal: True Independence for Teams, Features & Scaling

Allow **teams** to **develop**, **deploy** and **scale** their **specific business capabilities independently**

Ship new features faster **without risking the entire application**

Optimize resource usage by **scaling individual parts** of the system **based on their unique demand**

Solution: Microservices Architecture !!

What are Microservices?

Small, **independent** and **loosely coupled** services, each focused on a **specific business capability**

Owns **its codebase**

Managed by a **small**, **autonomous team**

Deployed independently

Communicates via **well-defined APIs** (often HTTP/gRPC)

Can **use** its **own technology stack** & **data store**

What is Microservices Architecture ?

[From Martin Fowlers Microservices article;]

*The microservice architectural style is an approach to developing a single application as a **suite of small services**, each running in **its own process** and **communicating** with lightweight mechanisms, often an **HTTP** or **gRPC API**.*

Decomposes a large, complex application into a collection of these small, independent services that work together

Organized around business capabilities, independently deployable, decentralized governance & data

Why to use Microservices Architecture

Agility & Faster Time-to-Market
Independent development & deployment of features

Flexible Scalability
Scale individual services based on specific needs

Small, Focused, Autonomous Teams
Increased productivity and ownership

Fault Isolation & Resilience
Failure in one service doesn't necessarily bring down the whole system

Technology Diversity
Teams can choose optimal tech for their service

Easier to Understand & Maintain
Smaller codebases per service

Challenges of Microservices Architecture

Increased Complexity (Overall System)
Managing many moving parts, inter-service com

Network Latency & Reliability
Communication is over the network, which is less reliable and slower

Distributed Data Management & Consistency
Ensuring data consistency across services is hard

Testing Complexity
E2E testing across multiple services is challenging

Deployment Complexity & Operational Overhead
Requires sophisticated automation

Debugging & Monitoring Distributed Systems
Tracing requests and pinpointing issues across services is harder

When to Use Microservices Architecture

Make Sure You Have a **'Really Good Reason**', focus on the **business problems**

Independent Scalability of services is **Critical**

High Agility & **Fast**, **Independent Deployments** are essential

Technology Diversity is a **Strategic Advantage**

You're Evolving a **Large**, **Complex Monolith** that's **Causing Significant Pain**

Autonomous, **Domain-Focused Teams** are a **Goal**

Best Practice: 'Monolith-First' approach

When NOT to Use Microservices Architecture

Small, **Simple Apps** or **Early-Stage Startups**

Lack of DevOps Culture & Automation

Inexperienced Teams (with Distributed Systems)

No Clear Business Driver for Distributed Complexity

You’re creating a "**Distributed Monolith" (Anti-Pattern!)**

Ignoring the **Need for Robust Infrastructure** & **Monitoring**

Using Shared Database across **microservices**

Why The Distributed Monolith Happen ?

Amazon Case: Rearchitecting from a Distributed Microservices to a Monolith

Video Streaming

Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%

The move from a distributed microservices architecture to a monolith application helped achieve higher scale, resilience, and reduce costs.

https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90

Key Questions to Ask Before Choosing Microservices

Is our current monolith truly the bottleneck for our business goals?

Have we exhausted options for optimizing/modularizing our monolith?

Do we have specific parts of the system that must scale independently or be deployed independently now?

Does our team have the skills and DevOps maturity for distributed systems?

Do the expected benefits (agility, scalability) clearly outweigh the significant operational complexity?

Architecture Comparison

Deployment Comparison

Monolithic

Microservices

What is the Database-per-Service Pattern?

Each **microservice owns** and **manages** its **own private database**

No other service can **directly access** another service's **database**

Services **interact with** each other's **data** only through **well-defined APIs** exposed by the owning service

Core Principle: Strong data encapsulation and true service autonomy

Why Use Database-per-Service?

True Loose Coupling & Service Autonomy

Independent Data Schema Evolution

Independent Scalability of Data Stores

Technology Flexibility (Polyglot Persistence)

Improved Fault Isolation

Clear Data Ownership & Boundaries

Polyglot Persistence - The Right Database for the Job

Using **multiple data storage** across **different microservices**

Choosing the **one best suited for each service's specific data model** and workload

Challenges of Database-per-Service

Joining Data Across Services
How to query or combine data that now lives in separate databases?

Distributed Transactions / Data Consistency
How to ensure consistency when a business operation spans multiple services?

Increased Operational Complexity
Managing multiple types and instances of databas

Way of Learning – The Course Flow

Problem: Monolith Bottlenecks at Scale

Business is Growing Rapidly -> Need for Speed!

Multiple Teams (Product, Sales, Payment) -> Need for Autonomy!

Demand for Immediate Feature Releases

Desire to Innovate & Experiment Quickly

Need for Independent Deployments

Requirement for Flexible, Granular Scaling

Goal: Handle millions of requests with acceptable latency

Implies: Not just a technology change, but an organizational shift

Solution: Microservices Architecture

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY, Separation of Concerns (SoC)

Single Responsibility Principle (SRP)

Database-per-Service Pattern

Polyglot Persistence

Non-FR (Update)

High Scalability
High Availability

Handle Millions of Concurrent Users/Requests

Independent Deployability

Technology Agnostic

Data Isolation

Resilience & Fault Isolation

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop with Initial Microservice Architecture

E-Shop's Development Technology Stack

Frontend (SPAs): Blazor, React, Angular, Vue.js

Backend Microservices:

Catalog Service: Node.js + MongoDB

Basket Service: .NET (C#) + Redis

Ordering Service: .NET (C#) + SQL Service

Identity Service: Python + PostgreSQL

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Microservices Architecture

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

6-eshop-microservices-first

Code: E-Shop Microservices Architecture

Way of Learning – The Course Flow

Evaluate: Microservices Architecture

Main Considerations

- Decomposition – Breaking Down Services
- Communications
- Data Management

- Transaction Management
- Deployments
- Resilience

Architecture Design – Vertical Considerations

Validating Our Service Boundaries

Our microservices are the right services? Are their boundaries ideal?

Rapid feature additions by specialized teams (Product, Sales, Payment)

Need for Flexible, Independent Scaling

Maximize agility, minimize inter-service dependencies

Problem: How to Decompose Microservices Effectively?

Are these services truly independent?

Do they encapsulate a coherent set of responsibilities?

Do their boundaries align with our business domain and team structures?

Are they an appropriate size?

Goal: Find Strategies for Decomposing Monoliths

Problems

Tightly coupled services

Chattiness Communication

Team Independency

Mismatched scaling units

Solutions

Microservices Decomposition Patterns

Gradual Migration Strategies

Microservices Decomposition Strategies

Microservices Decomposition Patterns

Breaking Down our EShop with Microservices Decomposition Patterns

Mehmet Ozkaya

Architecture Design Journey

Microservices Architecture Vertical Considerations

Why Decompose? The Drive for Independent Scalability

Primary Reason: Enable **different parts** of the application to **scale independently based on their unique demands**

Not all parts of an application have the **same scalability requirements**

"The Scale Cube" - Dimensions of Scaling

X-Axis (Horizontal Scaling / Scale Out)
Multiple identical copies of the apps behind a load balancer

Y-Axis (Functional Decomposition / Microservices)
Splits app by function/business capability into different services

Z-Axis (Data Partitioning / Sharding)
Each server instance handles a subset of data

Our Focus: Y-Axis Scaling through Functional Decomposition

Decomposing our **E-Shop** into **microservices** is an application of **Y-Axis scaling**

Independent, **smaller** and focused on **specific business functions**

Each function (now a service) to be **developed**, **deployed** and **scaled independently**

The Goal of Decomposition: Cohesion & Loose Coupling

Highly Cohesive: Implements a small set of strongly related functions. Does one thing well

Loosely Coupled: Interacts with other services through well-defined, stable APIs, hiding internal implementation

Decomposition Pattern: Decompose by Business Capability

Define services **corresponding** to **business capabilities**

A **business capability** is something a **business** does to **generate value**

Services are **stable units** as **capabilities change less frequently** than **processes**

Identified by **analyzing** the **organization's purpose**, **structure** and **business processes**

Decomposition Pattern: Decompose by Subdomain (DDD)

Define services corresponding to **Domain-Driven Design (DDD) subdomains**

DDD focuses on **modeling** the app's **'problem space'** – the **business domain**

A Domain consists of **multiple Subdomains**

Each Subdomain has its **own model** and **language** (**Ubiquitous Language**) within a **'Bounded Context'**

Services are **defined** around **these Bounded Contexts/Subdomains**

Domain-Driven Design (DDD) subdomains

Checklist for Evaluating Your Microservice Decomposition

Single Responsibility (SRP)
Does each service do "one thing" well, focused on a cohesive set of responsibilities?

Appropriate Size
Are services not too big (mini-monoliths) and not too small (nanoservices creating excessive chattiness)?

Communication Patterns
Is inter-service communication minimized? Avoid "chatty" interactions. If two services are constantly talking, should they be one?

Data Ownership & Isolation
Does each service own its data (Database-per-Service)? Is data duplication minimized or managed?

Checklist for Evaluating Your Microservice Decomposition2

Independent Deployability
Can each service truly be deployed without requiring changes or redeployments in other services? Are there hidden dependencies?

Team Autonomy
Can a small team independently develop, test, and deploy their service?

Alignment with Business
Do service boundaries make sense from a business capability or domain perspective?

E-Shop Functional Requirements & Use Cases

Functional Requirements

- List products
- Add products to the shopping cart
- Checkout the shopping cart and create an order
- List past orders and order item history

User Stories

- As a user, I want to list products
- As a user, I want to add products to the shopping cart
- As a user, I want to checkout the shopping cart and create an order
- As a user, I want to list my past orders and order history

Analysis EShop Domain - Nouns and Verbs

As a user I want to **list** **products**

As a user I want to be able to **filter** **products** as per **brand** and **categories**

As a user I want to **see** the supplier of **product** in the product detail screen with all characteristics of product

As a user I want to be able to **put** **products** that I want to **purchase** in to the **shopping cart** so I can check out

As a user I want to **see** the total cost all for all of the **items** that are in my **cart** so that I see if I can afford to buy

As a user I want to **see** the total cost of each **item** in the **shopping cart** so that I can re-check the price for items

As a user I want to be able to **specify** the **address** of where all of the products are going to be sent to

As a user I want to be able to **add** a note to the **delivery address** so that I can provide special instructions

As a user I want to be able to **specify** my **credit card** information during **check out** so I can pay for the items

As a user I want system to **tell** me how many items are in **stock** so that I know how many items I can purchase

As a user I want to **receive** **order** confirmation email with **order** number so that I have proof of purchase

As a user I want to **list** my old **orders** and **order items history**

As a user I want to **login** the system as a **user** and the system should remember my shopping cart items

Analysis E-Commerce Domain - Nouns

- Customer
- Product Catalog
- Shopping Cart
- Shopping Cart Items
- Order
- Order Details
- Supplier
- User
- Address
- Brand
- Category

Analysis E-Commerce Domain - Verbs

- List products applying to paging
- Filter products by brand, category and supplier
- See product all information in the details screen
- Put products in to the shopping cart
- See total cost for all of the items
- See total cost for each item
- Checkout order with purchase steps
- Specify delivery address
- Specify delivery note for delivery address
- Specify credit card information
- Pay for the items
- Tell me how many items are in stock
- Receive order confirmation email
- List the order and details history
- Login the system and remember the shopping cart items

Object Responsibility Diagram

Extracting Key Nouns - Potential Domain Objects

User Stories

- As a user, I want to list **products**
- As a user, I want to add **products** to the **shopping cart**
- As a user, I want to checkout the **shopping cart** and create an **order**
- As a user, I want to list my past **orders** and **order history**

Core Nouns: Product/Catalog, Shopping Cart, Shopping Cart Item, Order

Extended Nouns (as system grows): Customer, Order Details, Supplier, User Account, Address, Brand, Category

Initial Decomposition E-Shop into Potential Microservices

Product Catalog Service
Manages product information, categories, search

Shopping Cart Service
Manages users' active shopping carts

Ordering Service
Handles order creation, processing, and history

Customer/User Service
Manages user accounts, profiles, authentication

Order Fulfillment Process & Potential Further Decompose

E-Shop Microservices Landscape

What is the Strangler Fig Pattern?

Inspired by **strangler fig vines** that grow around and **eventually replace old trees**

A pattern for **incrementally migrating** a **legacy monolithic** system to **microservices**

Gradually create new microservices around the edges of the monolith

A **routing facade (Strangler Facade)** intercepts requests, **sending** them to either the **new microservice or** the **old monolith**

Over time, more **functionality** is **moved to new services** and the **monolith** is **"strangled" out**

Migrate to Microservices: EShop Modules to Microservices

The Strangler Fig Process in Action for EShop

Our Course's Design Focus Microservices

Assume this gradual Strangler Fig migration has been successfully applied to reach our designed microservices architecture

Our Course's Design Focus Microservices

https://github.com/aspnetrun/run-aspnetcore-microservices

Way of Learning – The Course Flow

Evaluate: Microservices Architecture

Main Considerations

- Decomposition – Breaking Down Services
- Communications
- Data Management

Our Decomposed Services Need to Work Together

Catalog, **Basket** and **Ordering** services are **separate**, but **business processes** often **span multiple services**

Example: Checkout might involve the Basket Service communicating with the Ordering Service

Unlike a **monolith**, they **can't just call methods** directly in memory

Problem: How to Communicate Microservices Effectively?

What communication protocols will they use (HTTP, gRPC, AMQP)?

What message formats will they exchange (JSON, XML, Protocol Buffers)?

How will services find each other on the network?

How will they handle API versioning as services evolve independently?

How do we design these service-to-service APIs to be efficient and avoid chattiness?

Goal: Well-Defined APIs & Communication Patterns

Goals

Clearly defined APIs for each service

Standardized communication protocols

Explore common synchronous communication using HTTP-based RESTful APIs

Solutions

Microservices Communication Patterns

Microservices Communications - The Basics

Microservices Communication Types - Synchronous or Asynchronous

RESTful API design for Microservices

GraphQL API flexible structured relational data for Microservices

gRPC high performance communication between internal Micros

Mehmet Ozkaya

Architecture Design Journey

Microservices Architecture Vertical Considerations

Problem: How to Communicate Microservices Effectively?

What communication protocols will they use (HTTP, gRPC, AMQP)?

What message formats will they exchange (JSON, XML, Protocol Buffers)?

How will services find each other on the network?

How will they handle API versioning as services evolve independently?

How do we design these service-to-service APIs to be efficient and avoid chattiness?

Communication Within a Monolithic Architecture

Primarily: **In-Process Communication**

Modules **reside** in the **same application process**

Communication = **Inter-Process Communication (IPC),** but more accurately, intra-process **method calls**

Benefits: Fast, Simple, No Network Overhead

Challenge: Leads to tight coupling, making independent scaling and deployment difficult

EShop Modular Monoliths
API
Modules
Catalog
Public API
Basket
Public API
Public API
Public API
Ordering
Identity
PostgreSQL

EShop Modular Monoliths
API
Modules
Catalog
Public API
Basket
Public API
Public API
Identity
PostgreSQL
Message Broker
Separated Service
Ordering

Module A
In-process
Method Calls
Module B - Public API
Public
API
Module B
Contracts
Module B

The Paradigm Shift: Communication in Microservices

Microservices are **distributed** – each runs in **its own process**, often on **different machines**

Direct method calls are **impossible**

Services must **interact inter-service communication protocols** over a **network**

This is a **major paradigm change** from monolithic design

Why and How Services Communicate

Microservices collaborate to **achieve broader business goals**

A **single user request** often **requires multiple services** to work together

Protocols for Sync Communication: HTTP with REST

For **synchronous request/response**, **HTTP** is a **widely used protocol**

REST (Representational State Transfer): Architectural style for **designing networked apps**, often built over **HTTP**

Uses **standard HTTP methods** (**GET, POST, PUT, DELETE**)

Resources **identified by URLs (/products**, **/orders/{id}**)

Typically **uses JSON** as the **data format** for **requests** and **responses**

Protocols for Sync Communication: HTTP with REST

HTTP Methods	
GET	Retrieve information
HEAD	Retrieve Resource Header
POST	Submit Data to the server
PUT	Save an object at the location
DELETE	Delete the object at the location
PATCH	Update single piece of data

Considerations for Network-Based Communication

Network Latency
Requests take time to travel over the network

Potential for Network Failures
Networks are not 100% reliable

Data Serialization/Deserialization
Objects must be converted to/from a format like JSON for network transmission

Security Concerns
Data over the wire needs protection

Service Discovery
How does Service A find the network address of Service B?

Microservices Communication Types – Sync or Async

Synchronous Communication

Client sends a **request** and **waits (blocks) for a response from** the **service**

The **client's thread** of **execution** is **paused until the response arrives** or a **timeout occurs**

Common Protocols: HTTP/HTTPS (for REST, GraphQL), gRPC

Think of it like **a direct phone call**: you ask and wait for an answer

Mehmet Ozkaya

Asynchronous Communication

Client **sends a message/event** and **does not wait for an immediate response**

The **client's thread** is **not blocked**; it **continues** its **own operations**

Often **involves** a **Message Broker** (e.g., RabbitMQ, Kafka)

Think of it like **sending an email** or **posting a letter**: you **send it** and **move on**

Mehmet Ozkaya

Request-Response vs. Event-Driven Communication

- Uses HTTP/REST, gRPC, GraphQL
- Direct calls, client expects an immediate result.
- Flow control is explicit (caller directs)
- Benefit: Simple, direct, immediate feedback
- Trade-off: Tighter coupling (temporal); caller blocked; risk of cascading failures if a downstream service is slow/down

- Uses Message Brokers (AMQP, Kafka)
- Producer emits events, consumers react
- Decoupled, no direct waiting
- Benefit: High decoupling, resilience, scalability
- Trade-off: More complex to trace/debug the overall flow; eventual consistency challenges

Choosing the Right Communication Style

Does the client need an immediate response to continue?

- **Yes -> "Consider Synchronous (e.g. HTTP/REST, gRPC)"**
- **No -> "Consider Asynchronous (e.g. Messaging/Events)"**

What is REST? (Representational State Transfer)

Designing **networked apps**, enabling **easy** and **fast client-server communication**

Introduced by **Roy Fielding** (2000) as an **alternative** to **SOAP/WSDL**

Runs on top of HTTP

Focuses on **transferring representations** of **resources**

Stateless, **Uniform Interface**, **Cacheable**, **Client-Server**, **Layered System**, **Code on Demand**

What are RESTful APIs? Applying REST Principles

Web services designed using **REST architectural principles** are called **RESTful APIs**

Communicate over **HTTP**, using standard **HTTP methods**

Resources are **identified** by **URIs**

Richardson Maturity Model

- **Level 0:** Define one URI, and all operations are POST requests to this URI
- **Level 1:** Create separate URIs for individual resources
- **Level 2:** Use HTTP methods to define operations on resources
- **Level 3:** Use hypermedia

Core HTTP Methods (Verbs) for Operations

HTTP Methods	
GET	Retrieve information
HEAD	Retrieve Resource Header
POST	Submit Data to the server
PUT	Save an object at the location
DELETE	Delete the object at the location
PATCH	Update single piece of data

RESTful API Design for Single Microservice

Rule1- REST APIs are designed around "resources"
The best practice is the resource URIs should be based on nouns = the resource and not verbs. REST API Url:

- https://e-shop.com/products -- Correct
- https://e-shop.com/create-product -- Wrong

Rule2- Every Resource has an identifier
Unique identifiers for that resource. REST API URL becomes for a particular product: https://e-shop.com/products/4

This will return single product which id = 4.

API frameworks route requests based on parameterized URI paths, so we can define a route for the path like **/products/{id}.**

RESTful API design for Single Microservice-2

Rule3- REST APIs are using JSON
REST API url has GET request, then it will return this response body as a JSON response:

- *{"productId":1,"name":"IPhone", "price":44.90,"category":"Electronics"}*

Rule4- REST APIs perform operations on Resources with HTTP verbs:
REST APIs built on HTTP protocol and perform operations on Resources with HTTP verbs, these verbs are GET, POST, PUT, PATCH, and DELETE.

When creating a product, we send HTTP POST request with contains product detail informations and return back to HTTP response that including 200 OK success response.

- ***Request:***
- *HTTP: POST*
- *Url: https://e-shop.com/products*
- *Body: {"productId":1,"name":"IPhone"}*
- *Response: 201 Created*

RESTful API Design for Relationship of Microservices

How can we design Rest API when you required to get a customer orders ?

- URL: https://e-shop.com/customers/6/orders

I want to retrieve products into order number 22 which ordered by customer 6:

- URL: https://e-shop.com/customers/6/orders/22/products --**WRONG**

This level of complexity can be difficult to maintain

Instead, we should keep URIs relatively simple

Resource URIs should not be more complex

than "**collection/item/collection**" path.

First we should get Customer orders with this API:

- URL: https://e-shop.com/customers/6/orders

After that we should locate the Order no 22 and get all products:

- URL: https://e-shop.com/orders/22/products

RESTful API Design with HTTP Methods

Resource	POST	GET	PUT	DELETE
/customers	Create new customer	Retrieve all customers	Update all customers	Remove all customers
/customers/1	N/A	Retrieve data for customer 1	Update data for customer 1 if exist	Remove customer 1
/customers/1/orders	Create new order for customer 1	Retrieve all orders for customer 1	Update all orders for customer 1	Remove all orders customer 1

RESTful API Path for Relational Data

Microservices don't share the same code base and **don't share data stores**. They communicate **through APIs** for data operations.

API versioning in Microservices RESTful APIs

If an **API changes**, there may broke some communications which depending on that API from other microservices

Services may required to add **new features** or **bug fixes** that require changing an existing API

API changes should be **backward compatible** and **not break** any communications

Common Strategies

URI Versioning: /api/v1/products, /api/v2/products

Header Versioning: Client requests a specific version via an HTTP header (X-API-Version: 2)

Query Parameter Versioning: /products?version=2

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY, Separation of Concerns (SoC)

Database-per-Service Pattern

Polyglot Persistence

Microservices Communication HTTP Based RESTful API design

Non-FR (Update)

High Scalability
High Availability

Handle Millions of Concurrent Users/Requests

Independent Deployability

Well-Defined, Discoverable APIs

Stateless API Interactions (REST principle)

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop Microservice with REST APIs

Design - E-Shop Microservice with REST APIs

Microservices	Resource	HTTP Methods	REST APIs	
Customer	**/customers**	HTTP GET HTTP POST HTTP DELETE	**GET /customers** **GET /customers/{id}** **POST /customers** **DELETE /customers/{id}**	
Product	**/products**	HTTP GET HTTP POST HTTP DELETE	**GET /products** **GET /products/{id}** **POST /products** **DELETE /products/{id}**	
Shopping Cart	**/carts**	HTTP GET HTTP POST HTTP DELETE	**GET /carts** **GET /carts/{id}** **POST /carts/{id}/checkout** **DELETE /carts/{id}**	
Ordering	**/orders**	HTTP GET HTTP POST HTTP	**GET /orders** **GET /orders/{id}**	

E-Shop's Development Technology Stack

Frontend (SPAs): Blazor, React, Angular, Vue.js

Backend Microservices:

C# ASP.NET, NodeJS, Java Spring Boot, Python Django

Catalog Service: ASP.NET Core Web API

Basket Service: ASP.NET Core Web API

Ordering Service: ASP.NET Core Web API

Identity Service: ASP.NET Core Web API

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Microservices Architecture

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

7-eshop-microservices-sync-call

Way of Learning – The Course Flow

The Challenge of Fetching Related Data with REST

Our business teams want richer UIs! Customer Dashboard

Customer's profile, A list of their recent orders

Details of products within each of those recent orders

The RESTful API Approach - Multiple Calls Needed

WebApp -> GET /customers/{id} -> Customer

WebApp -> GET /customers/{id}/orders -> Ordering

WebApp -> (Loop for each Order ID): GET /orders/{orderId}/products

Requires multiple, sequential API calls !!

Problem: N+1 Trap & Data Fetching Inefficiencies

N+1 Trap: Inefficient Data Fetching with Multiple REST API Calls for Related Data

Over-fetching: Getting more data than needed from some endpoints

Under-fetching: Not getting all related data in one go, leading to more calls (the N+1 problem)

Clients become **overly chatty**, leading to poor performance, and APIs might over-fetch or under-fetch data, leading to waste or more calls

Goal: More Flexible & Efficient Data Querying for Clients

Goals

Request exactly the data they need

Fetch complex, nested data involving multiple related entities

Reduce the number of round trips to the backend

Solutions

GraphQL API Design

Structural relational data with querying

GraphQL API

GraphQL: A Query Language for APIs

GraphQL is a **query language** for your **API**, and a **server-side runtime** for **executing those queries**

Developed by **Facebook (2012)**, publicly **released** in **2015**

Client-Driven: Allows **clients to ask for exactly the data they need**, nothing more, nothing less

Returns **predictable results** in the **same structure** as the **query**

Can **fetch data from multiple sources** in a **single request**

GraphQL

Key Characteristics of GraphQL

Ask for what you need, get exactly that
Clients control the data they receive, not the server. This avoids over-fetching and under-fetching

Get many resources in a single request
Fetch complex, nested data (a user, their posts, and their followers) in one trip

Evolve APIs without versioning
Add new fields and types without impacting existing client queries. Deprecate old fields gracefully

Core Concepts of GraphQL: Schema, Queries, Mutations, Resolvers

Schema

Defines all possible data (types, fields, relationship that clients can query. Acts as a contract

Queries

Requests sent by the client to fetch data. Structure mirrors the desired response

Mutations

Operations sent by the client to create, update, or delete data on the server

Resolvers

Responsible for fetching the data for specific fields in the schema. They connect the schema to your backend data sources

GraphQL

Example GraphQL Query & Response

orders with products

POST Request (Query in Body)

```
{
  orders {
    id
    totalAmount
    productsList {
      quantity
      product {
        name
        price
      }
    }
  }
}
```

JSON Response

```
{
  "data": {
    "orders": [
      {
        "id": 1,
        "totalAmount": 150,
        "productsList": [
          {
            "quantity": 100,
            "product": {
              "name": "orange",
              "price": 1.5
            }
          }
        ]
      }
    ]
  }
}
```

REST vs GraphQL APIs

Feature	REST	GraphQL
Data Fetching	Multiple Endpoints (Prone to over/under-fetching)	Single Endpoint Gets exact data needed
Client Control	Server-driven Server dictates response shape	Client-driven Client specifies response shape
Versioning	Coarse-grained /v1, /v2	Fine-grained Add/deprecate fields
Caching	Simple & Powerful Standard HTTP caching	Complex Requires custom/client-side logic
Complexity	Lower initial setup	Steeper learning curve

Fetching Data with REST and GraphQL

Blogging application, an application needs to display screen,

"**posts**" of the specific "**user**" and the last 3 "**follower**" names at the same screen.

users -> posts, users -> followers

How can we fetch this data with REST and GraphQL ?

Fetching Data with REST APIs

REST API, we have to call multiple endpoints to fetch data

1. **/users/<id>** endpoint to fetch the initial user data.
2. **/users/<id>/posts** endpoint that returns all the posts for a user.
3. **/users/<id>/followers** that returns a list of followers per user.

With REST, have to **make 3 requests** to different endpoints to fetch the required data

It is **over-fetching** since the endpoints return additional information that's not required

https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

Fetching Data with GraphQL APIs

GraphQL API, simply **send 1 single** query to the GraphQL server that includes the concrete data requirements

Then the server responds with a **JSON object** that includes required fields

Using GraphQL, the client can specify **exactly the data** it needs in a query

The structure of the server's response follows **precisely** the **nested structure** defined in the query

https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY, Separation of Concerns (SoC)

Database-per-Service Pattern

HTTP-Based RESTful API Design

GraphQL API Design

Non-FR (Update)

High Scalability
High Availability

Handle Millions of Concurrent Users/Requests

Independent Deployability

Efficient Data Fetching

Client-Driven Queries

Query aggregation

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop Microservice with GraphQL APIs

E-Shop's Development Technology Stack

Frontend (SPAs): Blazor, React, Angular, Vue.js

GraphQL API Layer:

Node.js with Express-GraphQL

ASP.NET Core with GraphQL .NET

Java with GraphQL Java

Way of Learning – The Course Flow

Designing Sync APIs for Microservices

There are 2 type APIs sync communication in microservices:

- Public APIs: API calls from the client apps
- Backend APIs: inter-service communication between backend services

Public APIs

Use RESTful APIs over HTTP protocol. RESTful APIs use JSON payloads for request-response, that easy to check payloads and easy agreement with clients

Backend APIs

Inter-service communication can result in a lot of network traffic. serialization speed and payload size become more important. Using gRPC is mandatory for increase network performance

The Scenario: High-Volume Inter-Service Traffic

Imagine our **E-Shop during a flash sale**

Many **users adding items** to **baskets simultaneously**

Basket Service **makes frequent internal API calls** for **each basket modification**

This generates **significant internal network traffic** using **REST/JSON**

Problem: Internal Communication Performance Bottlenecks

Problem: Network Overload - Default REST/JSON for **Internal Communication** is **Hitting Performance Limits**

For **high-throughput**, **low-latency internal service calls**, the overhead of **text-based JSON** and **standard HTTP/1.1** can **become** a **bottleneck**

Payload Size, Serialization/Deserialization, HTTP/1.1 Limitations, Real-time or Streaming Needs

Goal: More Performant Inter-Service Communication

Considerations

Low latency, high throughput

Small payload sizes, fast serialization

Strongly-typed contracts

Solutions

gRPC APIs scalable and fast APIs

Able to develop with different technologies

with RPC framework

gRPC: High-Performance RPC Framework

Open-source Remote Procedure Call (RPC) system by **Google**

Designed for efficient **inter-service communication** in **distributed systems**

Focuses on **high performance & low latency**

Uses **HTTP/2** for **transport** (**binary**, **multiplexing**)

Defines **service contracts** using **Protocol Buffers (Protobuf)**

Generates **cross-platform client/server code**

How gRPC Works - The RPC Model

1- .proto file defining a service ({ rpc DoSomething(Request) returns (Response); })

2- "gRPC Codegen" tool generating "Client Stub" and "Server Skeleton/Base"

3- "Client App" calls methods on "Client Stub."

4- "Client Stub" sends Protobuf message over HTTP/2 to "gRPC Server."

5- "gRPC Server" receives message, calls "Server Implementation"

6- Response flows back

When to Use gRPC? Ideal Use Cases

Internal Microservice-to-Microservice Communication
Especially when performance (low latency, high throughput) is critical

Polyglot Environments
Connecting services written in different languages

Network Constrained Environments
Where minimizing bandwidth usage is important due to binary payloads

Real-Time Applications requiring Streaming
Point-to-point streaming or bi-directional communication

NOT always a replacement for all REST APIs
Public-facing APIs consumed by browsers or diverse external clients often still benefit from the ubiquity and simplicity of REST/JSON

Example: gRPC in Our E-Shop's Inter-Service Calls

E-Shop Scenario: Basket Service needs product price & discount info for cart updates

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY, Separation of Concerns (SoC)

Database-per-Service Pattern

HTTP-Based RESTful API Design

gRPC API Design

Non-FR (Update)

High Scalability
High Availability

Handle Millions of Concurrent Users/Requests

Independent Deployability

High-Performance Internal Communication

Reduced Network Overhead

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop Microservice with gRPC APIs

E-Shop's Development Technology Stack

Frontend (SPAs): Blazor, React, Angular, Vue.js

Backend Microservices:

Public API Interface: .NET / ASP.NET Core Web API (RESTful)

Internal Service Interface: .NET / ASP.NET Core with gRPC Services

Evaluate: Microservice Architecture with gRPC APIs

Benefits

High-performance communication

Less bandwidth usage

Multi-language/platform supports

HTTP/2 and SSL/TLS usage

Supports Bi-directional Streaming operations

Drawbacks

Need additional library to generate codes that increases complexity

Can't read incoming-outgoing payloads that is not good for client to service communications

Way of Learning – The Course Flow

Microservices Synchronous Communications and Best Practices

Request/response communication with HTTP and REST Protocol (extends gRPC and GraphQL)

- **REST HTTP APIs** when exposing from microservices
- **GraphQL APIs** when structured flexible data in microservices
- **gRPC APIs** when communicate internal microservices

Question:

- How client applications can manage all these different protocols and communications inside of their applications ?

The Scenario: Client Try to Manage Direct Communications

Imagine our **E-Shop client application** (WebApp or Future Mobile App)

Know the **individual addresses** of **many different microservices**

Handle different communication protocols

Orchestrate calls to **multiple services** to **gather all the data** needed

Implement cross-cutting concerns

Problem: Direct Client-to-Service Communication

Problems

- Increased Client-Side Complexity
- Cause to chatty calls from client to service
- Hard to manage invocations from client app with all different protocols (HTTP, GraphQL, gRPC)
- Duplication of Effort Across Multiple Client Types
- Security & Cross-Cutting Concerns

Solutions

- Simpler, unified, and managed entry point
- Abstract away the complexity, consistent API
- **Microservices Communication Patterns Patterns (API Gateway, BFF)**

Microservices Communication Patterns - API Gateways

API Gateway Pattern

Gateway Routing Pattern

Gateway Aggregation Pattern

Gateway Offloading Pattern

Backends for Frontends Pattern BFF

Mehmet Ozkaya

Architecture Design Journey

Microservices Architecture Vertical Considerations

Problem: Direct Client-to-Service Communication

Problems

- Increased Client-Side Complexity
- Cause to chatty calls from client to service
- Hard to manage invocations from client app with all different protocols (HTTP, GraphQL, gRPC)
- Duplication of Effort Across Multiple Client Types
- Security & Cross-Cutting Concerns

Solutions

- Simpler, unified, and managed entry point
- Abstract away the complexity, consistent API
- **Microservices Communication Patterns Patterns (API Gateway, BFF)**

Communication Patterns - Starting with API Gateway

Our **primary focus** will be the **API Gateway pattern**

Provides a **single**, **unified entry point** for **all client requests**

Simplifies client code

Can **handle cross-cutting concerns**

Decouples clients from the **internal microservice structure**

The Solution - API Gateway Patterns

API Gateway Patterns:

- **Gateway Routing pattern**
- **Gateway Aggregation pattern**
- **Gateway Offloading pattern**

The Solution - API Gateway Patterns

Use **API Gateways** between client and internal microservices and it is **a single point of entry** to the client applications

API Gateway sits between **the client** and **multiple backend services** and manage **routing** to internal microservices

API Gateways can handle that **Cross-cutting concerns** like authentication, authorization, protocol translations, Rate Limiting, Logging, Monitoring, Load Balancing

API Gateway Patterns:

- **Gateway Routing pattern**
- **Gateway Aggregation pattern**
- **Gateway Offloading pattern**

The Solution - API Gateway Patterns

API Gateway Patterns:

- **Gateway Routing pattern**
- **Gateway Aggregation pattern**
- **Gateway Offloading pattern**

Gateway Routing Pattern

Route requests to **multiple services** by **exposing** a **unified set of endpoints** to the client

Client calls a **single gateway endpoint**

Gateway intelligently **routes** the **request** to the **appropriate** downstream service **based on path, headers, method, etc.**

Clients don't need to know the **addresses** or **specific paths** of **individual microservices**

Gateway Routing Pattern - Example

Gateway Routing Pattern - Benefits

Benefits: Decoupling Clients, Simple Client Endpoints, API Versioning & Blue/Green Deployments

Acts as an **intelligent reverse proxy using Layer 7** (application layer) routing

Gateway Aggregation Pattern

Consolidate multiple calls to **downstream microservices** into **a single client request**

Gateway **dispatches multiple individual requests** to the **relevant backend microservices**

Aggregates their **results into a single**, **consolidated response**

Single, **aggregated response** is then **sent back** to the **client**

Gateway Aggregation Pattern-2

Gateway Aggregation Pattern-2

Benefits: Reduces Chattiness and Latency, Simplifies Client Logic, Backend Abstraction

Doesn't become a bottleneck itself when performing aggregation

Gateway Offloading Pattern

Offload common, shared service functionalities from **individual microservices** to the API Gateway

SSL Termination: Handling HTTPS and decrypting traffic

Authentication & Authorization: Verifying user identities

Response Caching: Caching common responses

Rate Limiting and Throttling: Protecting backend services

Request Logging, **Tracing**, and **Correlation**

IP Allowlisting/Denylisting

Gateway Offloading Pattern -2

Summary: The Multifaceted API Gateway

API Gateway Pattern

A **single point of entry** for **all client requests** to the **backend microservices**

Sits between clients and the **microservices** ecosystem

Acts as a **reverse proxy**, **routing requests** to **appropriate services**

Can **aggregate responses** and **handle cross-cutting concerns**

Similar to the **Facade pattern** in object-oriented design, but **for distributed systems**

Why We Need an API Gateway

API Gateways directly **address** the **pain points** of **direct client-to-microservice communication**

Reverse Proxy and **Gateway Routing**

Request Aggregation (**Gateway Aggregation Pattern**)

Handling **Cross-Cutting Concerns** (**Gateway Offloading Pattern**)

Enabling **Protocol Translation** (e.g.client REST -> internal gRPC)

Routing	Authentication
Request Aggregation	Authorization
Service Discovery with Consul & Eureka	Throttling
Load Balancing	Logging, Tracing
Correlation Pass-Through	Headers/Query String Transformation
Quality of Service	Custom Middleware

A Typical Request Flow Through an API Gateway

Considerations & Potential Pitfalls with API Gateways

Single Point of Failure (SPOF) Risk
Requires high availability for the gateway itself

Potential Bottleneck
Must be properly scaled

Development & Management Complexity
It's another component to manage

Avoid "God Gateway" Anti-Pattern
Don't put excessive business logic in it

Consider Splitting for Diverse Clients (BFF)
A single gateway might become bloated

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

HTTP-Based REST, gRPC, GraphQL

Gateway Routing, Aggregation, Offloading Pattern

API Gateway Pattern

Non-FR (Update)

High Scalability High Availability

Handle Millions of Concurrent Users/Requests

Independent Deployability

Decoupled Clients from Backend

Centralized Cross-Cutting

Efficient Client Data Retrieval

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop Microservice with an API Gateway

E-Shop's Development Technology Stack

Frontend (SPAs): Blazor, React, Angular, Vue.js

API Gateway:

- **Open-source Gateway:** NGINX, Envoy
- **Cloud API Gateways:** Amazon API Gateway, Azure API Management and Google Cloud API Gateway
- **Framework-based Gateways:** Spring Cloud Gateway (Java) or Ocelot ,YARP (.NET)

Evaluating Microservices with an API Gateway

Benefits

Decouples Clients from Microservices

Reduced Client Chattiness

Centralized Cross-Cutting Concerns

Single Point of Entry

Potential for Protocol Translation

Drawbacks

Gateway Can Become a Bottleneck

Single Point of Failure (SPOF) Risk

Development & Management Overhead

Risk of "God Gateway" Anti-Pattern

Increased Latency

Way of Learning – The Course Flow

Focus: YARP API Gateway

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Microservices Architecture

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

8-eshop-microservices-yarp-api-gw

Way of Learning – The Course Flow

The Scenario: E-Shop on Multiple Fronts - Web & Mobile

Our E-Shop needs to serve different client experiences

Web Application: Rich UIs, more data displayed, powerful browsers

Mobile Application: Concise UIs, optimized for smaller screens, potentially different data needs, conscious of mobile data usage

Do our current microservice APIs efficiently serve both?

Problem: Client Apps has Different UI Requirements

Tailoring Data for **Diverse Clients** - **Generic APIs** Create **Client-Side Burdens**

A one-size-fits-all API from **backend microservices** often forces **client** applications to **do extra work** or **fetch unnecessary data**

Goal: Client-Specific Backend Logic & APIs

Problems

Frontend client apps has different UI and Data consumption requirements

Different data representation requirements need extra effort to adapt data for specific Client UI

Client codes complex and hard to maintain for future developments

Solutions

Client focused Interfaces

BFF-Backend for Frontend Pattern

Backends for Frontends Pattern - BFF Pattern

A **specialized API Gateway** for **each specific frontend** application or user experience

Each BFF provides an **API optimized** for its **particular frontend** (web, mobile, desktop...)

It **acts as a dedicated backend** service **for** a **single frontend**

Solves the "**one-size-fits-all**" API **problem**

Why and When to Use the BFF Pattern

Tailored Data & Experience
Each client gets data formatted and aggregated exactly as it needs for its UI/UX

Reduces Client-Side Logic
Moves data transformation, aggregation, and some presentation logic from the client to its BFF

Independent Evolution
Web BFF and Mobile BFF can evolve independently without impacting each other or the core backend services

Avoids "God" API Gateway
Prevents a single gateway from becoming overly complex and a bottleneck by trying to cater to everyone

Backends for Frontends Pattern - BFF Pattern

Considerations of BFF Pattern

Benefits

Optimized Client Experiences

Simplified Frontend Development

Independent Release Cycles for Frontends & their Backends

Clearer Separation of Concerns

Drawbacks

Increased Number of Backend Services to Manage

Potential Code Duplication Between BFFs

Added Network Latency

Operational Overhead

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

REST, gRPC, GraphQL

Gateway Routing, Aggregation, Offloading

API Gateway Pattern

Backends for Frontends (BFF) Pattern

Non-FR (Update)

High Scalability
High Availability

Handle Millions of Concurrent Users/Requests

Independent Deployability

Decoupled Clients from Backend

Optimized API for Each Client Type

Independent Evolution

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design - E-Shop Microservice with Dedicated BFFs

E-Shop's Development Technology Stack

Frontend (SPAs): Blazor, React, Angular, Vue.js

Web App BFF: .NET/ASP.NET Web API

Mobile App Client: iOS (Swift/Obj-C), Android (Kotlin/Java)

Mobile App BFF: .NET/ASP.NET Web API or Node.js, Go

Evaluating Microservices with an API Gateway

Benefits

Client-Focused Interfaces

Simplified Frontend Code

Improved User Experience

Independent Evolution

Avoids "God" API Gateway

Drawbacks

Increased Number of Deployable Units

Potential for Code Duplication

Added Network Hop

Operational Complexity

Service-to-Service Communications between Backend Internal Microservices

Created **API Gateways** and separate this API Gateways following the **BFF Pattern**

Sync request comes from the clients and goes to internal microservices **over the API Gateways**

What if the **client requests** are **required** to **visit more than one** internal **microservices ?** How we can manage internal microservice communications ?

Best practice is **reducing inter-service communication** as much as possible

In some cases, we can't reduce these internal communications due to operation **need to visit several internal services**

Client send query request to internal services to accumulate some data in the sceen page of frontend

Service-to-Service Communications between Backend Internal Microservices-2

User **addItem into shopping cart**, these operation need to get data from Product and Price microservices

Synchronous communcation over the API Gateways which is the Request/Response Messaging

The client **send http request** to the microservices for querying or adding item into Shopping Cart

Internal calls makes **coupling** each microservices

If one of the **microservices** is **down**, it can't return data to the client so **its not any fault-tolerance**

Service-to-Service Communications Chain Queries

If **service calls** are much then a few HTTP calls to multiple microservices, than it goes to **un-manageable** situation

Place Order Use Case: client checkout shopping cart, start order fulfillment processes

Request/Response Sync Messaging end up **with 6 sync chain HTTP Request**

Increase latency and **negatively impact** the performance, scalability, and availability

What if the **step 5 or 6 is failed ?**

It **negates** many of the **benefits** we seek from **microservices**

Way of Learning – The Course Flow

The Scenario: Composite View Requires Multiple Data

'Order Details' view for a user

Needs **data** from **Ordering Service**

Needs **data** from **Customer/User Service**

Needs **data** from **Product Catalog Service**

Needs **data** from a potential **Shipping Service**

How does the client or an initial service efficiently get this view?

Problem: Chained Query Problem with Direct Synchronous Calls

Problem: Inefficient Data Retrieval - Chained Synchronous Queries

Making **multiple sequential synchronous GET requests** to different services to build a **single view** is **inefficient** and **increase latency**

Goal: Efficient Data Aggregation for Queries

Problems

Multiple data retrieval operations

Complexity of how data is distributed across various microservices

Need single point for assembling composite data views

Solutions

Aggregate query operations

Service Aggregator Pattern

Service Aggregator, Service Registry/Discovery Patterns

Service Aggregator Pattern

Multiple backend service calls into a **single operation**, often for **specific client queries**

Receives a **single, composite request**

Dispatches requests to **multiple internal** microservices

Combines (aggregates) the results into a **single**, **coherent data structure**

Responds back with a **single, structured response**

Reduces chattiness and **client-side orchestration complexity**

The Challenge: Finding Services in a Dynamic Environment

Services are **independently deployed**

Instances scale up and **down dynamically**

Network locations (IP addresses, ports) are not **fixed**

Problem: How does one service (or an API Gateway/Aggregator) reliably find the current network location of another service it needs to call?

Solution: Service Registry and Discovery Pattern

What is the Service Registry and Discovery Pattern?

A **mechanism** for **services** to **register** their **locations** and for **other services** to **discover them**

A **centralized database** of **available service instances** and **their network locations**

Registration: When a microservice instance starts up it registers itself with the Service Registry

Discovery: Client queries the Service Registry by service name to get a list of available, healthy instances and their locations

There are **2 main service discovery patterns**:

- Client-side service discovery
- Server-side service discovery

Service Registry and Discovery Pattern

Service Discovery in Modern Cloud & Orchestration

Service Discovery is **often a built-in feature** of **container orchestrators** and **cloud platforms**

Kubernetes (K8s): Has built-in service discovery using DNS and Kube-Proxy. Services are exposed via stable internal DNS names

Cloud Platforms: Offer managed service registries and discovery mechanisms

.NET Aspire: Simplifies service discovery significantly in local development and provides abstractions for cloud deployment

Reduces the **need** to **implement** a **separate Service Registry infrastructure** from **scratch**

kubernetes

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

Gateway Routing, Aggregation, Offloading

API Gateway/ BFF Pattern

Service Aggregator

Registry/Discovery Pattern

Non-FR (Update)

High Scalability
High Availability

Handle Millions of Concurrent Users/Requests

Independent Deployability

Composite Data Retrieval

Dynamic Service Location Resolution

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ Service Aggregator/Registry

E-Shop's Development Technology Stack

Service Aggregator: .NET/ASP.NET Core Web API (can use HttpClientFactory to call other services, or gRPC clients)

Service Discovery: Platform Provided (Kubernetes DNS, .NET Aspire Service Discovery) or dedicated tools like Consul, Eureka

Other Tiers/Services: Remain as previously defined (SPA, Backend Microservices with their chosen stacks, Databases) – Lets Review !

HashiCorp Consul

Eureka

Adapt: Microservice Architecture with Service Aggregator, Registry/Discovery Pattern

Frontend SPAs

- Angular
- Vue
- React
- Blazor

API Gateways

- Kong Gateway
- Tyk API Gateway
- Express Gateway
- Amazon AWS API Gateway

Service Registry

- Netflix Eureka
- Kubernetes
- Serverless Orchestrators

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS

Database

- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair
- Postgres – Relational

Evaluate: Microservice Architecture with Service Aggregator, Registry/Discovery Pattern

Benefits

Aggregator

Reduces client-side chattiness and complexity

Abstracts complex backend data fetching

Centralizing logic into a specialized service

Registry/Discovery

Provide to register and discover microservices

Enables dynamic scaling and resilience

Keep track of microservice locations

Drawbacks

Increased Latency

Can become a new bottleneck

Another service to manage

Logic can be duplicated

HashiCorp Consul

Eureka

Way of Learning – The Course Flow

Service-to-Service Communications Chain Queries

If **service calls** are much then a few HTTP calls to multiple microservices, than it goes to **un-manageable** situation

Place Order Use Case: client checkout shopping cart, start order fulfillment processes

Request/Response Sync Messaging end up **with 6 sync chain HTTP Request**

Creates a **long chain** of **synchronous**, **blocking HTTP requests**

Service Aggregator doesn't solve the problem of how these backend services collaborate for a command-like operation

Problem: Long Running Operations Can't Handle with Sync Communication

Problems

Increased Latency & Poor Performance

Reduced Availability & Lack of Fault Tolerance

Tight Coupling Between Services

Resource Hogging

Best Practices

True Decoupling, Minimize communication between the internal microservices

Enhanced Resilience

Non-Blocking Operations

Solution

Asynchronous Message-Based Communications

Working with events

Microservices Asynchronous Communication & Event-driven Architecture

Benefits and Challenges of Asynchronous Communication

Asynchronous Message-Based Communication Types

Fan-Out Publish/Subscribe Messaging Pattern

Design E-Shop Microservices with Asynchronous Communication

Mehmet Ozkaya

Architecture Design Journey

Microservices Architecture Vertical Considerations

Problem: Long Running Operations Can't Handle with Sync Communication

Problems

Increased Latency & Poor Performance

Reduced Availability & Lack of Fault Tolerance

Tight Coupling Between Services

Resource Hogging

Best Practices

True Decoupling, Minimize communication between the internal microservices

Enhanced Resilience

Non-Blocking Operations

Solution

Asynchronous Message-Based Communications

Working with events

Microservices Communication Types – Sync or Async

What is Asynchronous Communication?

Producer **sends** a **request** (**event/message**) and **does not wait (non-blocking)** for an **immediate response**

The **producing service's thread** is **free** to **continue other work**

Typically involves a **Message Broker** as an **intermediary**, reliably **holds** the **event/message** until the **Consumer** is **ready to process**

Popular Protocol: **AMQP (Advanced Message Queuing Protocol)**

Communication happens via **events** or **messages**: **fire and forget**

The Role of the Message Broker & Event Bus

Message Brokers act as **intermediaries** like Event Bus in Event-Driven

Receive events/messages from **Producers**

Reliably store and **deliver** them to **interested Consumers**

Producers and **Consumers don't need to know** about each other directly or be available simultaneously

Can **support persistence**, **retries** and **different messaging patterns**

Mehmet Ozkaya

What We'll Cover in This Section

Benefits and Challenges of Asynchronous Communication

Asynchronous Communication Types (One-to-One Queues, One-to-Many Topics/Pub-Sub)

Event-Driven Architecture and the Publish/Subscribe Pattern

Designing & Evaluating our E-Shop with these asynchronous, event-driven patterns

Mehmet Ozkaya

Two Main Types of Asynchronous Communication

Single receiver message-based communication
one-to-one(queue) model or
Point-to-point model

Multi receiver message-based communication
one-to-many (topic) model or
publish/subscribe model

Single-receiver Communication (One-to-One / Queues)

One **producer sends** a **message**; one **consumer processes it**

Ideal for **delegating a specific task** or **command** to **another service** to be **processed**

Basket Service sends a **"CreateOrderCommand" message** to a **queue**

Message is **removed from** the **queue** after **successful processing** by the **consumer**

If the **consumer** is **down**, the **message remains in the queue** until the consumer is back

Common Technologies: RabbitMQ (queues), AWS SQS, Azure Service Bus Queues

Multiple-Receiver Communication (One-to-Many / Topics)

This is the **foundation** of **Event-Driven Architecture (EDA)**

One **producer publishes** an **event**; **multiple consumer services** can **subscribe**

Producer and **Consumers don't necessarily know each other**

Add **new subscriber** services to react to **existing events without changing the producer**

Messages persist in the **topic** for a **configured duration**

Apache Kafka, **RabbitMQ**, **AWS SNS**, **Azure Event Grid/Event Hubs**

Combining Communication Styles in a Microservices App

Often **use** a **combination** of **synchronous** and **asynchronous communication** styles

Synchronous (HTTP/REST, gRPC): For client queries needing immediate responses, or critical internal request/reply interactions

Asynchronous (Queues - One-to-One): For delegating tasks, commands, ensuring eventual processing

Asynchronous (Topics/Pub-Sub - One-to-Many / Event-Driven): For broadcasting state changes (events), decoupling services, enabling parallel processing by multiple interested parties

What is Event-Driven Architecture (EDA)?

An **architectural paradigm** where **system components react** to the **creation**, **consumption** and **reaction** to **events**

Service performs an **action** and then **produces** (**publishes**) an **event**

Other services that are **interested** in this **type** of **event** can **subscribe** to **it**

Typically **sent** to an **event bus** or **message broker**

Each **subscribed service** then **reacts** to the **event independently**, performing **its own logic**

Problem: Chain of request and highly coupled dependent microservices

The Publish/Subscribe (Pub/Sub) Pattern

Publishers: Services that generate and send events to a "topic" or "event bus" without knowing who (if anyone) is listening

Subscribers: Services that express interest in specific types of events from a topic and react when those events are received

Message Broker / Event Bus: The intermediary that manages topics and delivers events from publishers to all current subscribers of a topic

Publish/Subscribe Messaging Pattern-2

How Pub/Sub Enables Decoupling and EDA Benefits

Decoupling
Publishers and subscribers are completely independent

Scalability
Publishers, subscribers, and the message broker can often be scaled independently

Resilience
If a subscriber is temporarily down, events can often be queued or retained by the broker for later processing

Responsiveness
The publishing service can quickly fire an event and move on, not waiting for downstream processing

Considerations for Pub/Sub and EDA

Message Broker as a Critical Component
It can become a single point of failure if not made highly available

Debugging & Tracing Distributed Flows
Understanding the E2E flow of an operation triggered by an event can be complex

Eventual Consistency
Data consistency across services is achieved over tim not instantaneously

Message Ordering Guarantees
Not all brokers guarantee strict message order

Idempotent Consumers
Subscribers must often be designed to handle duplicate messages gracefully

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

API Gateway/ BFF Pattern

Event-Driven Architecture (EDA)

Publish/Subscribe Pattern (Fan-Out)

Non-FR (Update)

High Scalability High Availability

Handle Millions of ConcurrentRequest

Independent Deployability

High Decoupling between Services

Resilience & Fault Tolerance

Scalability of Event Processing

Eventual Consistency

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ Event Bus for Asynchronous Communication

E-Shop's Development Technology Stack

Message Broker / Event Bus: Kafka and RabbitMQ.

Cloud options: AWS SNS/SQS/EventBridge, Azure Service Bus/Event Grid, Google Pub/Sub

Microservice Architecture with Fan-Out Publish/Subscribe Messaging Pattern

Frontend SPAs

- Angular
- Vue
- React
- Blazor

API Gateways

- Kong Gateway
- Tyk API Gateway
- Express Gateway
- Amazon AWS API Gateway

Message Brokers

- Kafka
- RabbitMQ
- Amazon EventBridge, SNS

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS

Database

- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair
- Postgres – Relational

Evaluate: Microservice Architecture w/ Event Bus for Asynchronous Communication

Benefits

Extreme Decoupling

Improved Resilience & Fault Tolerance

Enhanced Scalability

Increased Agility

Better Responsiveness

Drawbacks

Increased System Complexity

Debugging & Tracing Distributed Event Flows

Eventual Consistency Challenges

Message Broker as a Critical Component (SPOF/Bottleneck)

Guaranteeing Message Order / Exactly-Once Processing

kafka

RabbitMQ

Way of Learning – The Course Flow

E-Shop Microservices with EDA & Publish/Subscribe

2 Events: Product Price Change & Basket Checkout

Eshop All Domain and Integration Events

E-Shop Microservices with EDA & Publish/Subscribe

Our "Code Exploration" Approach

Examine Codes

AppHost + Projects and codes, put break points

Run App

See Aspire Dashboard and graph -projects env variables

Test

E2E use cases see products, add basket, checkout, see orders

Observe

Logs, Traces, Db records, caches, buses

Code: E-Shop Microservices Architecture

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

9-eshop-microservices-async-rabbitmq

Revisiting Scalability Fundamentals

The ability of a system to **handle an increasing amount** of **work** (**requests, users, data**)

Measured by the **number** of **requests** an **application** can **successfully handle simultaneously**

When a **system can't handle** more, it has **reached** its **scalability limit**

Two Primary Scaling Strategies: Vertical vs. Horizontal

Vertical Scaling (Scale Up): Add more power (CPU, RAM, DISK) to an existing machine

Horizontal Scaling (Scale Out): Add more machines to your pool of resources

"The Scale Cube" - Dimensions of Scaling

X-Axis (Horizontal Scaling / Scale Out)
Multiple identical copies of the apps behind a load balancer **DONE!**

Y-Axis (Functional Decomposition / Microservices)
Splits app by function/business capability into different services **DONE!**

Z-Axis (Data Partitioning / Sharding)
Each server instance handles a subset of data

Our Focus: Z-Axis Scaling (Data Partitioning or Sharding)

X-Axis (Horizontal Scaling / Scale Out) : Run multiple Catalog ms instances

Y-Axis (Functional Decomposition / Microservices): Decomposing our monolith into services like Catalog, Order, and Basket

The Challenge: Horizontally Scaling Stateful Applications

If the server have a state or not ?

Stateless Services: Don't store client session data between requests

Stateful Services: Store data that persists and changes. Simply adding more copies creates challenges with data consistency, synchronization

How to Horizontally Scale Stateful Databases?

Our **microservices** are **mostly stateless** and **can scale out**

But their **dedicated databases** are **stateful!**

Vertical scaling of these **databases has limits** and **high costs**

Replicating traditional relational dbs for **writes creates major consistency challenges**

Way of Learning – The Course Flow

Reviewing Scalability of Our E-Shop Components
Can they scale?

Problem: Databases Create Bottlenecks When Scaling

Problems

Databases are inherently stateful

Traditional Vertical Scaling Has Limits

Provide Data Consistency Across Multiple Databases is Hard

Read/Write Bottlenecks, Connection Limits

Solution

Data management strategies

Handle high read and write loads

Scale horizontally more easily

Exploring different Database Types (SQL vs. NoSQL), Data Partitioning/Sharding and Advanced Replication/Clustering

Microservices Data Management - Choosing Right Database

Relational and NoSQL Databases

CAP Theorem – Consistency, Availability, Partition Tolerance

Data Partitioning: Horizontal, Vertical and Functional Data Partitioni

Database Sharding Pattern

Mehmet Ozkaya

Architecture Design Journey

Microservices Architecture Vertical Considerations

Microservices Data Management - Main Topics

Problem: Databases Create Bottlenecks When Scaling

Problems

Databases are inherently stateful

Traditional Vertical Scaling Has Limits

Provide Data Consistency Across Multiple Dbs is Hard

Read/Write Bottlenecks, Connection Limits

Solution

Scale Stateful Application Horizontal Scaling

Service and Data Partitioning along Business Boundaries - Shards/Pods

Use NoSQL Database to gain easy partitioning

How to Choose a Database for Microservices ?

Core Data Patterns & Principles for Microservices

Database-per-Service Pattern

Shared Database (Anti-Pattern)

API Composition Pattern

CQRS Pattern (separating reads and writes)

Event Sourcing Pattern

Saga Pattern (managing distributed transactions)

aya

The Database-per-Service Pattern

Each microservice owns and **manages** its own **private database**, **dedicated** to its **specific business capability**

Data is **encapsulated** within the **service boundary**

Other **services CANNOT directly access another service's database**

Services share or **retrieve data** from others **only through well-defined APIs** exposed by the owning service

Key Benefits of Database-per-Service

True Loose Coupling & Service Autonomy
Services evolve independently

Independent Data Schema Evolution
Schema changes in one service's DB don't break othe

Independent Scalability of Data Stores
Scale each DB based on its specific needs

Enables Polyglot Persistence
Choose the best database technology for each servic
unique data model and workload

Improved Fault Isolation
Failure in one DB primarily impacts its owning service.

Polyglot Persistence in Our E-Shop

Challenges of Database-per-Service Pattern

No Direct Cross-Service Database Joins
Querying data that spans multiple services requires alternative strategies

Distributed Transactions are Complex
Ensuring atomicity for operations that update data in multiple services requires patterns like Sagas

Operational Overhead
Managing multiple (potentially different types of) databases

Inter-Service Communication for Data Exchange
Services must rely on APIs to share data, introducing network latency and the need for resilient

The Shared Database Anti-Pattern

If we **don't follow Database-per-Service** pattern and use **Shared Database** for several microservices = **Anti-Pattern**

It is **against to microservices nature** and will cause serious problems in the future of apps

When using a **shared database**, the microservices **lose** their **core properties**: scalability, resilience, and independence

You will face to **develop big a few monolithic applications** instead of microservices

Shared database can **block microservices** due to **single-point-of-failure**

If shared database seems to be the best option for the microservices project, **should re-think** that we really need the microservices, the monolith would be the better choice

The Shared Database Anti-Pattern

The Shared Database Anti-Pattern - Pros and Cons

Benefits

Transaction management, that we don't need to span the transactions over the microservices

Decrease duplicate data. Since data is fully constrained, we can easily execute complicated queries with joins

Able to follow ACID. Consistency of data and state easily manage when process fails

Drawbacks

Microservices with shared databases can’t easily scale

Shared database will become a single point of failure

Microservices won’t be independent in terms of development and deployment

Polyglot Persistence

Microservices enables using **different kinds of data storing** technologies

Each development team can **choose the persistence technology** that suits the needs of their service best

Martin Fowler - **Polyglot persistence** will **come with** a **cost** - but it will come because the benefits are worth it

When **relational databases** are **used inappropriately**, they give damaged on app dev

Example of **looked up page elements by ID**, much better suited to a key-value No-SQL databases than relational databases

How to Choose a Database for Microservices ?

Database Choises - Relational and NoSQL Databases

Relational and **NoSQL** are **two types** of **database** systems commonly implemented in microservices architecture

Relational and **NoSQL** are different in terms of the **storing data**, and **accessing data**

Relational Databases - RDBMS

Stores data in **structured tables** with **pre-defined schemas** (columns with specific data types)

Uses **SQL (Structured Query Language)** for **data definition** and **manipulation**

Enforces **relationships** between **tables** using **primary** and **foreign keys**

Guarantees ACID properties (Atomicity, Consistency, Isolation, Durability) for **transactions**

Excellent choice for **highly structured**, **strong consistency**, **complex relational queries**

Examples: PostgreSQL, MySQL, SQL Server, Oracle

No-SQL Databases (Non-Relational Databases)

NoSQL (**'Not Only SQL'** or **'Non-Relational**') describes a wide variety of database technologies

Designed for **specific data models** and **flexible schemas**

Prioritize **scalability**, **performance** and **availability**

With **different consistency models** than **ACID** (e.g. **BASE** - Basically Available, Soft state, Eventual consistency)

Stores **data** in **formats** other **than relational tables** (e.g. **JSON documents**, **key-value pairs**)

No-SQL Document Databases

Stores **data** in **document format** (e.g. **JSON**, **BSON**)

Each **document** is **self-contained** and can have its **own structure** (**flexible schema**)

Good for **hierarchical data**; related data **often embedded** within a **single document**

Reduces the **need for complex joins**

Use Cases: Content management, product catalogs, user profiles

Examples: MongoDB, Couchbase, Amazon DocumentDB

No-SQL Key-Value Databases

Simplest NoSQL type: Stores **data** as a **collection** of **key-value pairs**

Each item **has** a **unique key** and a **corresponding value** (can be simple data or complex objects)

Extremely fast for **lookups by key**

Highly scalable and **performant** for **simple read/write** operations

Use Cases: Caching, user session management, real-time data lookups

Examples: Redis, Memcached, Amazon DynamoDB (can also be used as Document DB), Azure Cache for Redis

No-SQL Column-Based Databases

Stores data in **tables** with **rows** and **dynamic columns grouped** into **column families**

Optimized for **queries** over **large datasets** that **access specific columns**

Reads of **specific columns** are **very efficient** (avoids reading entire rows)

Highly scalable for **write-heavy workloads** and **Big Data** scenarios

Use Cases: Big Data analytics, time-series data, recommendation engines, high-volume logging

Examples: Apache Cassandra, Apache HBase, Google Bigtable, Azure Cosmos DB (Cassandra API)

Wide Column

No-SQL Graph-Based Databases

Stores data as **nodes** (**entities**) and **edges** (**relationships between entities**)

Designed to **efficiently store**, **query** and **traverse complex relationships**

Excellent for **data** where **connections** and **relationships** are as **important** as the **data itself**

Use Cases: Social networks, recommendation engines, fraud detection, knowledge graphs

Examples: Neo4j, Amazon Neptune, Azure Cosmos DB (Gremlin API)

When to Use Relational (SQL) Databases

Strong Data Consistency & ACID Compliance Required
Atomicity, Consistency, Isolation, Durability are critical for transactions

Data is Highly Structured & Predictable
Well-defined schema, clear relationships, data doesn't change structure frequently

Complex Queries & Joins are Needed
Leveraging the power of SQL for reporting or data analysis across related tables

Normalization is Important
Reducing data redundancy and ensuring data integrity

Workload Volume is Manageable for Vertical Scaling

When to Use NoSQL Databases

Flexible Schema & Dynamic Data is Required
Data structure evolves rapidly or varies significantly between items

High Scalability (Horizontal) is a Primary Concern
Need to handle massive read/write volumes beyond what single relational servers can manage

High Availability & Partition Tolerance are Critical
Designed for distributed environments

Specific Data Models are a Better Fit
Document, Key-Value, Graph, or Columnar data naturally fits the problem

Simplified Read/Write Patterns (Often No Joins)
When data is denormalized or accessed primarily by key

NoSQL Trade-offs: Consistency and Querying

Consistency Model (Often BASE - Eventual Consistency)
Many NoSQL DBs trade strong ACID consistency for higher availability and scalability

Querying Limitations (Compared to SQL)
Don't offer the full power of SQL for ad-hoc complex queries and joins

Transaction Support
Complex, multi-operation ACID-like transactions across different data items can be more challenging

Best Practices When Choosing a Data Store

Use the Right Tool for the Right Job
Don't default to one database type for everything

Embrace Polyglot Persistence
Leverage different database technologies based on specific service needs

Focus on the Data Type & Access Patterns
Match the database to how data is structured and used

Consider Consistency vs. Availability Trade-offs
Understand your service's requirements

Align with Transactional Boundaries

Factor in Team Competence

Problem: Databases Create Bottlenecks When Scaling

Problems

Databases are inherently stateful

Traditional Vertical Scaling Has Limits

Provide Data Consistency Across Multiple Dbs is Hard

Read/Write Bottlenecks, Connection Limits

Solution

Scale Stateful Application Horizontal Scaling

Service and Data Partitioning along Business Boundaries - Shards/Pods

Use NoSQL Database to gain easy partitioning

How to Choose a Database for Microservices ?

A Question Set to Guide Database Selection

1. **Data Consistency Level** Needed? (Strict/ACID vs. Eventual)
2. **Schema Flexibility**? (Fixed vs. Dynamic/Flexible Schema)
3. **Data Volume** & **Workload Type**? (Low/High Volume, Read-Heavy, Write-Heavy)
4. **Query Complexity**? (Simple Lookups vs. Complex Joins/Relational Queries)
5. **Deployment Structure**? (Centralized vs. Geo-Distributed)
6. **Performance Needs**? (Low Latency Requirements)
7. **Scalability Needs**? (Vertical and/or Horizontal)
8. **Availability Requirements**? (How much downtime is acceptable?)

The "Impossible Triangle" & Introduction to CAP Theorem

Can one database provide ALL
High Scalability, High Availability, Low Latency, High Performance, Complex Joins, AND Strong ACID Consistency?

Unfortunately, **NO**

Why?

Because of **fundamental trade-offs** in **distributed syste**
explained by the **CAP Theorem**

Understanding the CAP Theorem – C, A, P

CAP Theorem (Eric Brewer, 1998): In a distributed data store, you can only guarantee two out of three properties

C – Consistency
Every read receives the most recent write or an error. All nodes see the same data at the same time

A – Availability
Every request receives a response, without guarantee that it contains the most recent write. The system is always operational

P – Partition Tolerance
The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

CAP Theorem: The Trade-off (CP vs. AP Systems)

In a distributed system, network partitions (P) are a fact of life. Therefore, you must choose between C and A

CP Systems (Choose Consistency over Availability)
Sacrifice availability during a partition to ensure all accessible data is consistent

AP Systems (Choose Availability over Consistency)
Sacrifice strong consistency during a partition to ensure the system always responds

Most modern large-scale distributed systems (especially NoSQL) lean towards AP, optimizing for availability and handling eventual consistency

Way of Learning – The Course Flow

The Limitations of a Single Database Server Instance

We have **Database-per-Service**, which is **great** for **autonomy**

BUT, if each service's database **runs** on a **single server instance**...

...that **single instance** can still **become** a **bottleneck** for its **specific service** as **data** and **load grow**

Vertical scaling (bigger server) helps, but **has limits** and **high costs**

Problem: Single Database Server Performance Degradation

The **Single Server Squeeze** - When **One Database Instance Isn't Enough**

As **data volume** and **request loads** grow for a specific **microservice**, its **single database server instance** can **suffer** from **severe performance degradation**, **resource exhaustion**, and **manageability issues**

Single database server has **physical limits, becomes primary bottleneck**

Goal: Distributing Data Within a Service's Database

Problems

DBs become very slow by time with app grows

Single database goes un-manageble situation

Database operations are become slower

Goals

Distribute DB data across multiple database

Query performance by working with smaller data subsets

Scalability and availability at the data tier

Solutions

Data Partitioning Strategies

Horizontal, Vertical, and Functional Data Partitioning

What is Data Partitioning?

The **process** of **dividing a large database** into **smaller**, **more manageable** and **independent pieces** called **partitions**

Each partition can often be **managed** and **accessed separately**

Goal: Distribute data to **improve availability**, **scalability** and **performance**

Often transparent to **client applications** (they see a **single logical database**)

Why Use Data Partitioning?

Scalability: Distribute data and load across multiple servers (horizontal scaling for data)

Availability: Failure of one partition/server doesn' necessarily affect others

Performance: Queries can operate on smaller data subsets; parallel processing across partitions

Manageability: Smaller partitions are easier to back up, index and maintain

Security: Isolate sensitive data in specific partitions

Three Main Data Partitioning Strategies

1. **Horizontal Partitioning (Sharding):** Divides a table by rows
2. **Vertical Partitioning:** Divides a table by columns
3. **Functional Partitioning:** Divides data by business function or bounded context

Horizontal Partitioning - Sharding

Divides a **table** into **multiple smaller tables** (**shards**) by **distributing its rows**

Each shard has the **same schema** (**columns**) but **contains** a **different subset** of the **total rows**

Data is **distributed based on** a **Partition Key** (CustomerID, ProductID, Region)

Client queries use the **partition key** to **locate** the **correct shard**

Benefit: Spreads load and data across servers, excellent for scaling high-volume tables

Challenge: Choosing a good partition key is critical to avoid "hot spots" (one shard getting too much load)

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Vertical Shards

VS1

CUSTOMER ID	FIRST NAME	LAST NAME
1	Alice	Anderson
2	Bob	Best
3	Carrie	Conway
4	David	Doe

VS2

CUSTOMER ID	CITY
1	Austin
2	Boston
3	Chicago
4	Denver

Horizontal Shards

HS1

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston

HS2

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
3	Carrie	Conway	Chicago
4	David	Doe	Denver

An Overview of Sharding & Partitioning | Hazelcast

Vertical Partitioning – Row Splitting

Divides a **table** by **distributing** its **columns** into **different partitions** (new tables)

Each partition holds a **subset of the original table's columns**, but **all rows** (linked by a common key)

Columns are **often divided based on** their **access patterns** or **characteristics**

Frequently accessed columns in **one partition**, large **text/blob columns** or **less frequently accessed** columns **in another**

Benefit: Improves query performance if queries often access only a subset of columns; can separate frequently changing data from static data

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Vertical Shards

VS1

CUSTOMER ID	FIRST NAME	LAST NAME
1	Alice	Anderson
2	Bob	Best
3	Carrie	Conway
4	David	Doe

VS2

CUSTOMER ID	CITY
1	Austin
2	Boston
3	Chicago
4	Denver

Horizontal Shards

HS1

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston

HS2

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
3	Carrie	Conway	Chicago
4	David	Doe	Denver

An Overview of Sharding & Partitioning | Hazelcast

Functional Partitioning (Domain/Context-Based)

Divides data based on how it's used by different **business functions** or **bounded contexts** within app

Data is **segregated** according to its **usage domain**

Aligns closely with **how we decompose microservices** by **business capability** or **DDD subdomains**

Example: Product data in one set of tables/schema, Order data in another, Customer data in yet another.

This is often a **natural outcome** of the **Database-per-Service pattern** & **Polyglot Persistence**

What is Database Sharding?

Sharding is a specific type of **Horizontal Data Partitioning**

Divides a **database** into **smaller**, **faster**, more **manageable** pieces called **shards** (or chunks)

Each shard is an **independent database** instance, holding a **distinct subset** of the **total data rows**, but all **shards share** the **same schema**

A **Shard Key** (a column or set of columns) **determines which shard** a **particular row** of **data belongs** to

A **routing mechanism** (**sharding logic**) **directs** application **queries** to the **appropriate shard**

https://www.redhat.com/architect/pros-and-cons-sharding

Why Shard? Benefits of the Database Sharding Pattern

Massive Scalability (Horizontal)
Scale your data tier by adding more shards as data grows. Overcomes single server hardware limits

Higher Availability & Fault Isolation
Failure of one shard doesn't necessarily affect other shards; data in other shards remains accessible

Geographic Distribution
Shards can be located closer to users in different regions, reducing latency (GeoSharding)

https://www.redhat.com/architect/pros-and-cons-sharding

Tinder - A Real-World Example of Database Sharding

Tinder: A popular **location-based dating app**

Core Requirement: Quickly find and match users who are geographically close to each other (e.g. within ~160km)

Challenge: Millions of users globally, massive number of potential matches, location is key

Solution Approach: GeoSharding – Location-based database sharding

How Tinder Uses GeoSharding

Tinder segments its **user database** based on **geographic location**

The **world map** is conceptually **divided** into **regions** (**shards**)

Each **user's profile data** is **stored** in a **shard corresponding** to their **current** or **primary location**

When a **user searches** for **potential matches**, the **query** is **primarily directed** to the **shard**(s) **relevant** to their **location** and **search radius**

Benefits for Tinder: Drastically Reduced Search Scope, Lower Latency, Scalability

Cassandra No-Sql Database
Peer-to-Peer Distributed Wide Column Database

Cassandra is very good example of implementation **Database Sharding Pattern** and using **Horizontal Scaling** features

Cassandra is a **distributed database** from Apache Foundation

It is **highly scalable** and designed to manage very **large amounts of structured data**

It provides **high availability** with **no single point of failure**

Apache Cassandra is a highly scalable, high-performance **distributed database** designed to **handle large amounts** of data across many **different located servers**

Apache Cassandra feautures;

- Elastic scalability - It is scalable, fault-tolerant, and consistent
- No single point of failure
- Column-oriented database
- Flexible data storage
- Easy data distribution

COUNTRY	CITY	POPULATION
USA	New York	8.000.000
USA	Los Angeles	4.000.000
FR	Paris	2.230.000
DE	Berlin	3.350.000
UK	London	9.200.000
AU	Sydney	4.900.000
DE	Nuremberg	500.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
FR	Toulouse	1.100.000
JP	Tokyo	37.430.000
IN	Mumbai	20.200.000

Partition Key

https://cassandra.apache.org/_/cassandra-basics.html

Cassandra Architecture

https://cassandra.apache.org/_/cassandra-basics.html

Why Cassandra No-Sql Database ?

Cassandra has an **auto-sharding feature** and it helps keep data divided among nodes.

All of the columns that will **decide on which node** the data should be kept are called "**Partition Key**"

Partition Key is set to **Sensor#** and **Date**

The data will be **stored in nodes** determined by inserting the value of these **two columns** with hash function. It becomes **Partition Keys**

CAP Theorem: either Availability or Consistency must be selected in a distributed system

Microservice architectures decided that **Availability is more important** to them and sacrifice for strict consistent and **embrace to Eventual Consistency** would be sufficient

https://cassandra.apache.org/_/cassandra-basics.html

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

Shared DB Anti-pattern

Polyglot Persistence

CAP Theorem

Data Partitioning

Database Sharding Pattern

Non-FR (Update)

High Scalability High Availability

Handle Millions of Concurrent Request

Independent Deployability

Scalability for Specific DBs Stores

High Availability & Fault Tolerance DBs

High Load for Data Operations

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ Database Sharding Pattern

E-Shop's Development Technology Stack

Relational Databases

PostgreSQL, SQL Server

NoSQL Databases (Natively Distributed)

Apache Cassandra – NoSQL Peer-to-Peer

MongoDB – NoSQL Document DB

Redis – NoSQL Key-Value Pair

Cloud NoSQL Databases

Amazon DynamoDB, Azure Cosmos DB

Amazon DynamoDB

cassandra

Microservice Architecture with Database Sharding Pattern – Cassandra

Frontend SPAs
- Angular
- Vue
- React

API Gateways
- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Message Brokers
- Kafka
- RabbitMQ
- Amazon EventBridge, SNS

Backend Microservices
- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS

No-SQL Database
- Cassandra – NoSQL Peer-to-Peer
- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair

Cloud No-SQL Database
- Amazon DynamoDB
- Azure CosmosDB

Evaluate: Microservice Architecture w/ Database Sharding

Benefits

Massive Horizontal Scalability for Data

Improved Query Performance

Higher Availability & Fault Isolation

Better Data Manageability

Master-Master (Master-less) architecture with Ceph

Distributed Wide Column Database

Drawbacks

Increased Architectural Complexity

Cross-Shard Queries & Joins

Transactions Spanning Shards

Hot Spots

Operational Overhead

Complex JOIN requires operations

Amazon DynamoDB

cassandra

Way of Learning – The Course Flow

Evaluating Our Sharded, Distributed Data Architecture

Database-per-Service: Each service owns its data

Polyglot Persistence: Using optimal DB tech per service

Database Sharding: Distributes data and load for individual services like Ordering

Questions

How do we query data lives in different services/shards?

How do we handle write commands that logically span these boundaries?

How do we optimize read and write paths separately at scale?

Problem: Cross-Service Queries and Write Commands on Distributed Scaled Databases

Considerations

Cross-services queries that retrieve data from several microservices ?

Separate read and write operations at scale ?

Problems

Cross-Service Queries with Complex JOIN operations

Read and write operations at scale

Distributed Transaction Management

Solutions

Microservices Data Query Pattern and Best Practices

Materialized View Pattern

CQRS Design Pattern

Event Sourcing Pattern

Microservices Data Management - Commands and Queries

Microservices Data Query Pattern and Best Practices

Materialized View Pattern

CQRS Design Pattern

Event Sourcing Pattern

Eventual Consistency Principle

Mehmet Ozkaya

Architecture Design Journey

Microservices Architecture Vertical Considerations

Microservices Data Management - Main Topics

Problem: Cross-Service Queries and Write Commands on Distributed Scaled Databases

Considerations

Cross-services queries that retrieve data from several microservices ?

Separate read and write operations at scale ?

Problems

Cross-Service Queries with Complex JOIN operations

Read and write operations at scale

Distributed Transaction Management

Solutions

Microservices Data Query Pattern and Best Practices

Materialized View Pattern

CQRS Design Pattern

Event Sourcing Pattern

Core Data Patterns for Managing Commands & Queries

Microservices Data Query Best Practices

Materialized View Pattern

CQRS (Command Query Responsibility Segregation) Design Pattern

Event Sourcing Pattern

Eventual Consistency Principle

aya

How can we manage these cross-services queries ?

Direct HTTP Communication
Not a good solution that makes coupling each microservices, and loose power of microservices independency

Async Communication
The best practice is reducing inter-service communication as much as possible and use async communication.

Client **send query** request to **internal microservices** to **accumulate** some **data**

Those query request wait **immediate response** so we can't proceed with async communication

Transient errors, **Network congestion** or any overloaded microservice can result in long-running and failed operations

Materialized View Pattern
Reduce inter-service communication and provide sync response

Microservices Cross-Service Queries
Example Use Case – Get Item with Details

Microservices Cross-Service Queries
Chain Queries – Get Order with All Details

Problem: Inefficient Cross-Service Queries for Reads

Problems

Cross-Service Queries with Complex JOINs

Return Sync Response with low latency

Provide loosely coupling with decouple services

Reduce inter-service communication

Considerations

Sync Communication: Use Service Aggregator Pattern but it increase coupling and latency

Async Communication: Provide decoupling but query request are waiting immediate response

Solutions

Materialized View Pattern

CQRS Design Pattern

What is the Materialized View Pattern?

A service stores its own local, denormalized, and optimized read-only copy of data that it needs from other services

This local copy is the "**Materialized View**" or "**Read Model**"

Data is pre-computed and stored in a format optimized for the service's query needs

Eliminates the need for expensive, synchronous cross-service calls for reads

Benefits of Using Materialized Views

Query Performance & Reduced Latency

Reduced Inter-Service Coupling

Data Optimized for Queries

Reduced Load on Source Services

Key Considerations for Materialized Views

Data Duplication
You now have copies of data in multiple places. This is a strategic duplication

Data Synchronization & Staleness
How and when is the materialized view updated when the original source data changes?

Data Ownership
The original service still owns the source-of-truth data

Updating Materialized Views - Asynchronous Synchronization

Event-Driven Updates (Publish/Subscribe)

The service owning the source data publishes an event when its data changes

The service holding the materialized view subscribes to these events and updates its local copy accordingly

This leads to eventual consistency for the materialized view

Other methods: Scheduled batch updates, triggers

CQRS — Command Query Responsibility Segregation

Separates operations that **change state (Commands)** from operations that **read state (Queries)**

Often involves using **different data models** and potentially **different data stores** for **reads** and **writes**

Goal: Maximize performance, scalability, and security for each path independently

Why CQRS? Different Needs for Reads and Writes

Reads and **writes** often have very **different characteristics** and **requirements**

Writes (Commands)
Need strong consistency (ACID), involve business logic/validations, may be less frequent but more complex

Reads (Queries)
Much higher volume, need low latency, may require joining data from multiple entities

A single data model optimized for **one** can be **suboptimal** for the **other**

CQRS – Read and Write Operations

CQRS enables '**separation of concerns**' for data handling

Write Side (Commands)
Uses a model and often a data store optimized for updates, validations, and transactional consistency

Read Side (Queries)
Uses a model and often a separate data store optimized for queries

Data Synchronization
The read store is updated (asynchronously via events) when changes occur in the write store, leading to eventual consistency

CQRS – Read and Write Operations

Benefits and Drawbacks of CQRS

Benefits

Independent Scaling

Optimized Data Models

Improved Query Performance

Increased Security & Simpler Logic

Drawbacks

Increased Complexity

Eventual Consistency

Code Duplication

Synchronization Logic

Best Practices for CQRS & Real-World Example (Instagram)

Best Practices

Physically separate read and write databases

Optimize read-side schema for queries

Use different database technologies

Example: Instagram Architecture

Uses PostgreSQL (relational) for core user data, writes, and strong consistency

Uses Cassandra (NoSQL) for massive scale user feeds, activity streams

user, media, friendship etc

How to Sync Read and Write Databases in CQRS ?

Commands update the **Write Database** (**source of truth**)

Queries read from the **Read Database** (**optimized read model/materialized view**)

Problem: How do **changes** in the **Write Database propagate** to the **Read Database** to **prevent stale data** for **queries**?

Direct, **synchronous updates** are often **not feasible** or desirable as they **negate** some **CQRS benefits**

Solution: Event-Driven Synchronization via Asynchronous

The most common approach: **Event-Driven Architecture**

When **Write Database modified**, app **publishes** an **event detailing** the **change**

This **event** is **sent** to a **Message Broker / Event Bus**

Event consumers subscribe to these **events** and **update** the corresponding **Read Database(s) / Materialized View(s)**

This **synchronization** is **asynchronous**

Implications of Asynchronous Synchronization: Eventual Consistency

Asynchronous synchronization leads to **Eventual Consistency**

Read Database will **eventually become consistent** with the **Write Database**

There might be a **brief period** of **latency** where the **read model doesn't yet reflect** the **latest writes**

Fundamental **trade-off** in **distributed systems** fo **achieving scalability** and **decoupling**

App and **UI** must be **designed** to **handle** or **tolerate** this **potential** (usually **short**) **staleness**

Other Synchronization Considerations & Related Patterns

Database Replication
Some read models can be simple read replicas of the write database

Change Data Capture (CDC)
CDC can monitor the write database's transaction log and publish change events directly, decoupling event publishing from app code

Event Sourcing
If using Event Sourcing for the write side (storing all changes as a stream of events), these same events are naturally perfect for building and updating read models / materialized views

What is the Event Sourcing Pattern?

Instead of **storing** only the **current state** of an **entity**, **Event Sourcing persists every state change** as an **immutable event**

Each event represents a **fact**, **something** that **happened** (OrderCreated, ProductPriceChanged, ItemAddedToCart)

These **events** are **stored sequentially** in an **append-only log** called an **Event Store**

The **Event Store** becomes the ultimate **source of truth**

How Event Sourcing Works?

The **current state** of an **entity** is **derived by replaying** its **events** from the **Event Store**

Events from the **Event Store published** to ar **Event Bus** to **notify other parts** of the syster to **update read models**

Provides a **full audit log** inherently

Enables temporal queries (what was the sta at a specific point in time?)

Event Store **becomes the heart of your data persistence** for **state-changing operations**

CQRS with Event Sourcing

CQRS with Event Sourcing

Write Side (Commands)

Instead of updating a stateful record, new events are generated and appended to the Event Store, The Event Store is the write model

Read Side (Queries)

Materialized Views (Read Models) are created and updated by consuming the events published from the Event Store

CQRS with Event Sourcing

Eventual Consistency Principle

When using **asynchronous updates** from an **Event Store** (Write Model) to **Read Models**, **Eventual Consistency** is the norm

These **events** are then **asynchronously processed** to **update** the **Read Models**

Changes (**events**) are written to the **Event Store** (**source of truth**)

There will be a **short delay** before the **Read Models reflect** the **latest state**

The system **eventually becomes consistent** across all views

Microservices Data Patterns and Principles

Materialized View Pattern

CQRS Design Pattern

Event Sourcing Pattern

Eventual Consistency Principle

Instagram - A Global Scale Application

Instagram operates at a massive global scale

Hundreds of **millions** of **daily active users**

Billions of **interactions** (likes, shares, stories viewed)

Huge volumes of **data** (photos, videos, user information, feeds)

Challenge: Handling this load requires a robust, distributed and highly scalable architecture

Multiple data centers worldwide to serve users **with low latency**

Simplified Instagram System Architecture Overview

Web/Application Servers (running **Django**)

Caching layers (**Memcached**)

Asynchronous Task Queues (**RabbitMQ**)

And crucially, a **mix of database** technologies

Key Database Choices: Primarily PostgreSQL (Relational) and Apache Cassandra (NoSQL)

Users connect to the **nearest data center**, following **CAP Theorem**

Availability and **Partition Tolerance** are more critical than immediate, strong **Consistency**

Instagram's Use of PostgreSQL (Relational)

PostgreSQL: For data requiring strong consistency, relational integrity, and complex transactions

Architecture: Master-Slave replication. All writes go to the Master. Reads can be distributed to slaves

Data Stored: User information (profiles, credentials), User relationships (friendships, followers)

Ensures data integrity for core user data and metadata. **Master** provides a **single source** of **truth** for **writes**

Instagram's Use of Cassandra (NoSQL Wide-Column)

Cassandra: For massive scale, high write throughput and high availability for read-heavy, eventually consistent data

Architecture: Peer-to-Peer (Masterless/Multi-Master). Every node can handle reads and writes

Data Stored: User Feeds, Activity Feeds/Stories, Counters, Direct Messaging

Extreme horizontal scalability, **fault tolerance** (no single point of failure), optimized for write-heavy and read-heavy workloads where **eventual consistency** is **acceptable** (Built-in sharding)

Cassandra Architecture

https://cassandra.apache.org/_/cassandra-basics.html

Key Architectural Principles in Instagram's Data Design

Polyglot Persistence

CQRS (Command Query Responsibility Segregation)

Database-per-Service

CAP Theorem in Action

Multi-Dimensional Scaling

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture, EDA

Patterns &Principles

KISS, YAGNI, DRY

Database-per-Service Pattern

Polyglot Persistence

CAP Theorem

CQRS, Event Sourcing, Materialized View Pattern

Eventual Consistency

Non-FR (Update)

High Scalability High Availability

Handle Millions of ConcurrentRequest

Independent Deployability

High Write Throughput & Auditability

Optimized Read Performance

Scalability of Read and Write Paths

Resilience in Data

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ CQRS & Event Sourcing

E-Shop's Development Technology Stack

Event Store (Write DB)

PostgreSQL, SQL Server or for EventStore

EventStoreDB, Apache Kafka (as an event log)

Read Models (Read DBs)

Apache Cassandra – NoSQL Peer-to-Peer

MongoDB – NoSQL Document DB

Redis – NoSQL Key-Value Pair

Cloud NoSQL Databases

Amazon DynamoDB, Azure Cosmos DB

Amazon DynamoDB

cassandra

Microservice Architecture with CQRS, Event Sourcing, Eventual Consistency – FULL VIEW

Frontend SPAs
- Angular
- Vue
- React

API Gateways
- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Message Brokers
- Kafka
- RabbitMQ
- Amazon EventBridge, SNS

Backend Microservices
- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS

READ DB - No-SQL Database
- Cassandra – NoSQL Peer-to-Peer
- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair
- Amazon DynamoDB, Azure CosmosDB

WRITE DB - Relational Database
- PostgreSQL
- SQL Server
- Oracle

Evaluating CQRS with Event Sourcing

Benefits

Optimized & Scalable Read/Write Paths

Full Audit Trail & Historization

Temporal Queries Possible

Flexible Read Models

Improved Performance

Drawbacks

Increased System Complexity

Eventual Consistency for Read Models

Steeper Learning Curve

Snapshotting for Performance

Evolving Event Schemas

Amazon DynamoDB

Way of Learning – The Course Flow

Where We Are: Scaled, Optimized Services... But Distributed Data

Database-per-Service: Each service owns its data

Polyglot Persistence: Using optimal DB tech per service and sharding

CQRS/Event Sourcing: Optimized read/write paths, full audit via events

Questions

What happens when a single business operation (Place Order) requires coordinated, consistent updates across multiple of these independent data stores?

Problem: Manage Distributed Transactions and Consistency between Microservices

Considerations

Distributed Transactions that required to visit several microservices ?

Consistency across multiple microservices ?

Rollback transaction and run compensating steps ?

Problems

Distributed Transaction Management

Rollback Transaction on Distributed Environment

Run Compensate Steps if one of service fail

Solutions

Saga Pattern for Distributed Transactions

Transactional Outbox Pattern

Compensating Transaction pattern

CDC - Change Data Capture

Microservices Distributed Transactions

Saga Pattern for Distributed Transactions - Choreography and Orchestration

Transactional Outbox Pattern

Compensating Transaction pattern

Dual-Write Problem

CDC - Change Data Capture

Mehmet Ozkaya

Architecture Design Journey

Microservices Data Management - Main Topics

- Polyglot persistence
- The Database-per-Service
- The Shared Database Anti-Pattern

- The API Composition
- The CQRS pattern
- The Event Sourcing Pattern

- The Saga Pattern
- Transactional Outbox
- Compensating Transactions
- CDC - Change Data Capture

Microservices Architecture Vertical Considerations

Problem: Manage Distributed Transactions and Consistency between Microservices

Questions

How to perform operations that span multiple services (e.g. Place Order)?

How to achieve data consistency across these services?

How to handle failures and rollbacks in a distributed environment?

Problems

Traditional single-database ACID transactions don't easily apply across microservices

Solutions

Saga Pattern for Distributed Transactions

Transactional Outbox Pattern

Compensating Transaction pattern

CDC - Change Data Capture

Core Data Patterns for Managing Distributed Transactions

Saga Pattern (Choreography & Orchestration based)

Transactional Outbox Pattern

Compensating Transaction Pattern

Understanding the Dual-Write Problem

Change Data Capture (CDC)

Mehmet Ozkaya

Transaction Fundamentals & ACID Principles

What is a **Transaction** (in a traditional, single database)?

A **sequence** of **database operations** treated as a **single logical unit of work**

All-or-Nothing: Either all operations complete successfully, or if any fails, the entire sequence is rolled back, leaving the database in its original consistent state

Managed by **ACID Principles**

The ACID Principles Explained

ACID guarantees reliability of **database transactions**

A (Atomicity): All or Nothing

C (Consistency): DB remains valid

I (Isolation): Concurrent transactions don't inte

D (Durability): Committed changes persist

Transaction Management in Monolithic Architectures

All operations within a **business transaction** typically **occur against** a **single database**

ACID properties (Atomicity, Consistency, Isolation, Durability) are **provided** by the **relational database**

Easy to **commit** all **changes** or **rollback everything** if one step fails

Example: place_order() in E-Shop – payment, stock, order update all in one ACID transaction

```
function place_order()
    do_payment
    decrease_stock
    send_shipment
    generate_bill
    update_order
```

Microservices Shift: Why Distributed Transactions are Hard

Each service has **its own database** (Database-per-Service)

A **single business transaction** involves **coordinated updates across multiple, independent databases**

Traditional ACID transactions (two-phase commit - 2PC) across **multiple databases** are **complex**, **slow**, **reduce availability** and **create tight coupling**. They are generally **NOT recommended** for **microservices**

We **need new patterns** to **manage consistency**

How to achieve Data Consistency across multiple service ?

The Bigger Challenge: Multi-Step Write Transactions

'Place Order' which **involve writes** to **multiple services**?

Atomicity: Either **ALL steps succeed OR NONE of them** effectively **do** (if one fails, previous steps must be undone/compensated)

Eventual consistency alone **isn't enough** for the **atomicity** of the **business transaction** itself. We **need** a way to **manage** the **overall workflow**

CAP Theorem & Eventual Consistency for Distributed Transactions

CAP: In distributed systems, we **often choose Availability & Partition Tolerance**, accepting **Eventual Consistency**

Most **large-scale microservice** applications **prioritize** being **highly available** and **scalable**

This means for many **distributed operations**, **immediate**, **strong consistency** across all services **is sacrificed**

The **patterns** we'll learn for **distributed transactions** (like Sagas) are built around **achieving overall business consistency** eventually

Recapping Bounded Contexts - Defining Service Boundaries

Complex domains are **broken into Subdomains**

A **Bounded Context** provides a **logical boundary** around a **specific subdomain** or a group of **closely related subdomains**

Within a **Bounded Context**, models and language are **consistent** and **unambiguous**

Microservices are often **designed** to **align** with these **Bounded Contexts**

DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.

http://martinfowler.com/bliki/BoundedContext.html

From Service Boundaries to Transactional Boundaries

Customer/User, **Product Catalog**, **Shopping Cart**, **Discount**, **Ordering**

While **each service handles** its **local transactions** within its **own database**

Many **E2E business use cases** require **coordinated** actions **across multiple services** and their **Bounded Contexts**

This is where we define **Microservices Transactional Boundaries**

What are Microservices Transactional Boundaries?

A **business operation** needs to **achieve** an **'all-or-nothing' outcome** across **multiple**, **independent microservices**

It's about **maintaining overall business consistency** when data is distributed

We **cannot use traditional ACID** transactions across these service boundaries

We aim for **eventual consistency** for the overall **process**, using patterns to **coordinate local transactions**

Identifying these boundaries helps us **understand** the **minimum set of services** that **must coordinate** for a given **business outcome**

E-Shop "Order Fulfillment" Transactional Boundary

1. **Customer** checkouts Shopping Cart
2. **Order** is created in Ordering Service
3. **Payment** is validated by Payment Service
4. **Stock** is deducted by Inventory Service
5. **Shipment** is arranged by Shipment Service
6. **Notification** is sent by Notification Service

(**All services proceed local ACID transactio**

Problem: How to **coordinate** these **local transactions** to **achieve overall consistency**?

Microservices Transactional Boundaries
Order Fulfillment – Success

Microservices Transactional Boundaries
Order Fulfillment – Rollback

What is the Saga Pattern?

A **Saga** is a **sequence** of **local transactions distributed across multiple microservices**

Each local transaction updates data **within** a **single service** (using its own ACID)

When a **local transaction completes**, it **triggers** the **next local transaction**

If any local transaction fails: The saga executes compensating transactions to undo the work done by preceding successful local transactions

Manages long-running, **multi-step business transactions** in a distributed system

Saga Implementations: Choreography vs. Orchestration

There are **two type of saga** implementation ways:

Choreography-based SAGA Implementation

Orchestration-based SAGA Implementation

Choreography-based SAGA Implementation

Each microservice communicates with the other microservices by **exchanging events** using a message broker

Event-based approach allows for a **more decentralized** and **flexible** way to implement the SAGA pattern, as each microservice can **publish to events** to the others **in real-time**

Choreography provides to **coordinate sagas** with applying **publish-subscribe principles**

Each microservices run its **own local transaction**, **publishes events** to **message broker** system and **trigger local transactions** in other microservices

Order Fullfilment with Choreography-based SAGA Implementation

Order Fullfilment Rollback with Choreography-based SAGA Implementation

SAGA pattern provides a way to **roll back** the **changes** made by each microservice

If the Inventory microservice encountered an error while reserving the items in the customer's order, it could **publish a failure event** to the message broker

Order fulfillment microservice would then **execute a compensating transaction** to undo the charges to the customer's payment method and cancel the order

Benefits of Choreography-based SAGA

Decentralized and flexible
By using an event-based approach, each microservice can react to events published with the others to coordinate their actions. This allows for a more decentralized and flexible approach to implementing the SAGA pattern.

Decouple direct dependency of microservices when managing transactions.

Avoid Single Point of Failure
Since there is no orchestrator, responsibilities are distributed across the saga participants.

Simple workflows
This way is good for simple workflows, if they don't require too much microservices transaction steps.

Drawbacks of Choreography-based SAGA

More complex to manage
Each microservice communicates via events with the others, the choreography-based approach can be more complex to manage and may require more coordination among the microservices

SAGA workflow become confusing when adding new steps into flow

Cyclic Event Consume Risk
There's a Cyclic Event Consume Risk dependency between saga participants because they have to consume each other's commands

Result
Implementing the SAGA pattern will depend on the specific needs and constraints of the distributed system. It may be a good fit for some systems, but may not be suitable for others

Choreography-based implementation of SAGA is good for simple workflows if they don't require too much micorservices transaction steps

Orchestration-based SAGA Implementation

Involves using a **central orchestrator** service to **coordinate** and **manage** the **individual sagas** or microservices that make up a transaction

The orchestrator is responsible for **initiating the transaction** and **ensuring** that each **saga performs** its step in the **correct order**

If any of the sagas fail to complete their step, the **orchestrator** can use the **compensating transactions** to **roll back** the **changes** and **restore** the system to its **original state**

Orchestration provides to **coordinate sagas** with a **centralized controller** microservice that orchestrate the saga workflow and invoke to **execute local** microservices **transactions** in **sequentially**

Orchestration-based SAGA Implementation - Rollback

Benefits and Drawback of Orchestration-based SAGA Implementation

Benefits

Provides a clear and centralized point of control for managing transactions

Make easier to understand and debug the system, and add new transactions or modify existing

Drawbacks

The orchestrator can also become a single point of failure, and if it goes down, the entire system may be unable to complete transactions

The orchestrator can become a bottleneck if the system is heavily loaded, as all transactions must go through it

Result

It can be a useful approach for managing transactions in a distributed system, but it is important to carefully consider the trade-offs and potential drawbacks

Orchestration way is good for complex workflows which includes lots of steps

But this makes single point-of-failure with centralized controller microservices and need implementation of complex steps

What is a Compensating Transaction?

A **transaction** specifically designed to **reverse** the **effects** of a **previously successfully completed local transaction** within a **Saga**

It's the "**undo" operation** for a specific step in a **distributed business process**

Triggered when a **subsequent step** in a **Saga fails**, requiring the **overall business operation** to be rolled back

Goal: To achieve an overall "**all-or-nothing**" outcome for the **business transaction** by ensuring that if the entire **Saga cannot complete**, the system is **returned** to a **consistent state** (from a business perspective)

Key Characteristics of Compensating Transactions

Specific to an Operation
Each local transaction in a Saga should have a corresponding compensating transaction

Idempotent by Design
Compensating transactions should be designed to be idempotent

Can Also Fail
A compensating transaction is also a distributed operation and could potentially fail

Achieves Eventual Consistency
The process of compensation brings the system back to a consistent state eventually

EShop Order Fulfillment - Compensating Transactions

Implementing Compensating Transactions - Challenges

Complexity

Data for Compensation

Idempotency is Key

Potential for Failure

Order of Execution

What is the Dual Write Problem?

Occurs when an **app needs** to **atomically update data** in **two different external systems** (e.g. a **database** and a **message broker**) as part of **a single logical operation**

Persisting a state change to **a database** and **publishing** an **event** to a **message broker**

The **'dual write'** refers to these **two separate write operations**

Each operations has its **own local transaction** or **acknowledgment mechanism**

Dual Write: Creating an Order & Publishing an Event

What can go wrong ? - Dual Write Problems

Scenario 1: Write to Database, then Publish Event
What if the app crashes or the event publish fails after the database commit? -> Inconsistent state! (Order exists, but no event)

Scenario 2: Publish Event, then Write to Database
What if the event is published before the database commit, and the DB commit then fails? -> Inconsistent state! (Event sent for an order that doesn't exist)

Goal: Reliable Atomic Operations Across DB & Broker

We need **Database change** AND the **publishing** of a corresponding **event** happen **atomically**

Avoid losing events if the DB write succeeds but publish fails

Avoid publishing events if the DB write fails

Solutions

Transactional Outbox Pattern

CDC - Change Data Capture

Best Practice

Use Red Hat **Apache Kafka** and **CDC** using **Debezium** in event-driven applications

Use New databases like **CockroachDB** which has built-in **Change Data Capture** feature

What is the Transactional Outbox Pattern?

Reliably publish events by **persisting** them **within the same local database transaction** as the **business state changes**

Instead of **directly publishing** an **event** to a **message broker** after a DB write, the **event** is **first written** to an "**Outbox" table within** the **same database** as the business data

This **write** to the **business table** and the **Outbox table** happens in **a single, atomic local database transaction**

How the Transactional Outbox Pattern Works

Step 1 (Atomic Commit)

Business logic updates domain entities

An event representing the change is created

Both the entity changes AND the event are saved to their respective tables within a single local database transaction

Step 2 (Asynchronous Publishing)

A separate process monitors the Outbox table

It reads unprocessed events from the Outbox

It reliably publishes these events to the Message Broker

Marks the event in the Outbox as processed (or deletes it)

Customer Places an Order Use Case
Transactional Outbox pattern in Microservices

Step 1 (Atomic Commit)

Step 2 (Asynchronous Publishing)

Why Use the Transactional Outbox Pattern?

Solves the Dual Write Problem
Guarantees that if a business transaction commits, its corresponding event will eventually be published

Increased Reliability for Event Publishing
Decouples event publishing from the main transaction flow

Improved Performance of Business Transactions
The main transaction only involves fast, local DB writes; publishing to an external broker is deferred

Data Consistency
Ensures the event published accurately reflects the committed state of the database

Considerations and Implementation Details

Message Relay Implementation
Can be a background worker, a separate small service, or leverage Change Data Capture (CDC) on the Outbox table

At-Least-Once Delivery & Idempotent Consumers
Downstream consumers must be idempotent to handle potential duplicate events

Eventual Consistency
Events are published after the main transaction commits and by a separate process, so there's a delay

Outbox Table Bloat/Cleanup
Need a strategy for cleaning up processed events from the Outbox table

Way of Learning – The Course Flow

Recapping the Transactional Outbox Flow

Step 1 (Atomic Commit)

Step 2 (Asynchronous Publishing)

Problem: Inefficiencies of Polling the Outbox Table

Problems

Wasted Resources & Performance Impact

Latency in Event Publication

Potential for "Thundering Herd" on the Outbox

Difficulty in Choosing Polling Interval

Risk of New Dual Write (Relay to Broker)

Solutions

Change Data Capture (CDC) with the Outbox Pattern

What is Change Data Capture (CDC)?

Set of **software design patterns** used to **determine** and **track data** that **has changed** so that **action** can be **taken using** the **changed data**

Captures Inserts, **Updates** and **Deletes** from a source database

Makes these **changes available** for **consumption** by other systems **in near real-time**

Typically works by **monitoring** the **database's transaction log** (**commit log**)

Avoids inefficient polling of **data tables**

CDC with the Transactional Outbox Pattern

CDC is an **excellent way** to **implement** the **'Message Relay'** for the **Outbox Pattern**

Microservice writes **business data AND** an **event** to its **Outbox Table** in **one local transaction**

A **CDC tool monitors** the **database transaction log** for new **rows inserted** into the **Outbox Table**

When a **new event record** is **committed** to the **Outbox Table**, **CDC captures** it and **publishes** it to the **Message Broker**

Eliminates polling: **Near real-time**, **low-overhead** event detection

Example: E-Shop Order Creation with CDC & Outbox

CDC - Change Data Capture with Outbox Pattern EShop Create Order Use Case

Id	AggregateType	AggregateId	Type	Payload
ec6e	Order	123	OrderCreated	{ "id" : 123, ... }
8af8	Order	456	OrderDetailCanceled	{ "id" : 456, ... }
890b	Customer	789	InvoiceCreated	{ "id" : 789, ... }

Outbox Table

https://debezium.io/blog/2019/02/19/reliable-microservices-data-exchange-with-the-outbox-pattern/

Native CDC Support in Modern & Cloud Databases

Many **modern databases** offer **built-in** or **enhanced CDC capabilities**

CockroachDB

Azure Cosmos DB

Amazon DynamoDB Streams

These **features** can **simplify implementing** the **Outbox pattern** or other **event-driven integratio**

CockroachDB for CDC and Outbox Pattern

Distributed SQL DB with a built-in CDC feature (changefeeds) that can stream changes to Kafka, etc

https://www.cockroachlabs.com/docs/stable/change-data-capture-overview.html

Azure Cosmos DB for CDC and Outbox Pattern

Offers a "Change Feed" feature, allowing applications to listen for and react to item changes in a container

https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/change-feed-design-patterns

Amazon DynamoDB Streams for CDC and Outbox Pattern

Captures item-level modifications in DynamoDB tables and streams these changes in near real-time. Can trigger AWS Lambda functions

https://medium.com/aws-lambda-serverless-developer-guide-with-hands/dynamodb-streams-using-aws-lambda-to-process-dynamodb-streams-for-change-data-capture-2e3ab8df27ca

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture, EDA

Patterns &Principles

Database-per-Service

CAP Theorem, Database Sharding Pattern

Saga Pattern (Choreography/Orchestration)

Transactional Outbox Pattern

Compensating Transaction

Change Data Capture (CDC)

Non-FR (Update)

High Scalability High Availability

Independent Deployability

Data Consistency Across Services Reliable Completion or Compensation of Business Transactions

Fault Tolerance in Multi-Step Processes

Avoid Dual-Write

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ Saga, Outbox & CDC

E-Shop's Development Technology Stack

Message Brokers / Event Bus (for Saga)
Apache Kafka, RabbitMQ, AWS EventBridge/SNS/SQS, Azure Service Bus

CDC Tools
Debezium (for PostgreSQL, MySQL, SQL Server, MongoDB), Oracle GoldenGate

Databases with Built-in CDC/Change Feeds
CockroachDB, Azure Cosmos DB, Amazon DynamoDB Streams

kafka debezium

Amazon DynamoDB

Microservice Architecture with SAGA, Transactional Outbox and CDC Pattern

Frontend SPAs

- Angular
- Vue
- React

API Gateways

- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Message Brokers Event Bus

- Kafka
- RabbitMQ
- Amazon EventBridge, SNS

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS

Log-based built-in CDC Database

- CockroachDB
- Azure CosmosDB
- Amazon DynamoDB Streams

CDC – Open Source Tool

- Debezium

Evaluate: Microservice Architecture w/ Sagas, Outbox, CDC

Benefits

Reliable Distributed Transactions

Reliable Event Publishing (Outbox + CDC)

Highly Decoupled Services

Improved Fault Tolerance & Resilience

Scalability

Drawbacks

Increased System Complexity

Eventual Consistency Challenges

Debugging & Tracing Distributed Sagas

Compensating Transaction Logic

Operational Overhead

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Microservices Architecture

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

11-eshop-microservices-outbox

Way of Learning – The Course Flow

Where We Are: Robust & Consistent, But Is It Always Fast?

Reliable event publishing (Outbox/CDC)

Strategies for distributed transactions (Sagas)

Optimized read models (CQRS/Materialized Views)

Questions

What if repeated access to the source-of-truth databases or frequently re-computed data by individual services is still too slow for certain operations, especially under load?

Problem: The Cost of Frequent Data Store Access

Problems

Database Operations Can Be Expensive

Repetitive Access to the Same Data

Latency in Event-Driven Flows

Performance of Synchronous API Calls

Considerations

Reduce latency for common read operations

Decrease load on primary data stores

Improve overall system responsiveness and throughput

Solutions

Distributed Caching

Storing frequently accessed data in a distributed cache

Microservices Distributed Caching

Microservices Distributed Caching Patterns and Practices

Cache-Aside Pattern

Caching strategies

Cache invalidation

Cache consistency

Mehmet Ozkaya

Architecture Design Journey

Microservices Data Management - Main Topics

- Polyglot persistence
- The Database-per-Service
- The Shared Database Anti-Pattern

- The API Composition
- The CQRS pattern
- The Event Sourcing Pattern

- The Saga Pattern
- Transactional Outbox
- Compensating Transactions
- CDC - Change Data Capture

- Store frequently accessed data

Problem: The Cost of Frequent Data Store Access

Problems

Database Operations Can Be Expensive

Repetitive Access to the Same Data

Latency in Event-Driven Flows

Performance of Synchronous API Calls

Considerations

Reduce latency for common read operations

Decrease load on primary data stores

Improve overall system responsiveness and throughput

Solutions

Distributed Caching

Storing frequently accessed data in a distributed cache

What is Caching?

Storing frequently accessed data in a **temporary**, **high-speed storage** location (the **'cache'**) to serve **future requests faster**

Goal: Reduce expensive operations (DB queries, network calls, complex computations)

Benefits: Increased **performance** (lower **latency**), better **scalability** (**reduced load** on source), improved **availability**

Mehmet Ozkaya

Common Types of Caching Architectures

In-Memory Cache
Data stored in the application's own memory

Disk Cache
Data stored on disk. Slower than in-memory, but persistent

Distributed Cache
External cache service shared by multiple application instances

Why Distributed Caching for Microservices?

In **microservices**, services are often **scaled horizontally** (**multiple instances**)

In-memory cache per **instance leads** to **data inconsistency** and **cache misses**

Distributed Cache provides a **shared**, **consistent caching** layer accessible by **all service instances**

Improves performance of **individual** services by **reducing** calls to their **primary DBs** or other services

Allows the **cache** itself to be **scaled independently**

Benefits of Distributed Caching in a Microservices

Improved Performance & Reduced Latency
Faster data retrieval from shared in-memory cache

Reduced Load on Backend Data Stores
Protects databases from excessive read traffic

Enhanced Scalability
Both services and the cache can scale independently

Increased System Resilience
Services can potentially operate with cached data if a primary DB is temporarily slow

Consistent Data View (across service instances)
All instances of a service share the same cache

Common Caching Strategies

How your **app interacts** with the **cache** and the **primary data store defines** the **caching strategy**

Goal: Reduce latency, decrease database load, save costs

Choice depends on: Data characteristics (size, access frequency, update frequency) and app needs

Common Strategies

Cache-Aside (Lazy Loading)

Read-Through

Write-Through

Write-Back (Write-Behind)

Caching Strategies in Distributed Caching

Cache-Aside Strategy (Lazy Loading)
Receives a read request, App checks Cache, Cache Hit: Returns data from Cache, Cache Miss: App reads data from Primary Database

Read-Through Strategy
Makes a read request directly to the Cache, Cache checks if data is present, Cache Hit: Cache returns data to App, Cache Miss: Cache is responsible for fetching the data from DB

Write-Through Strategy
Performs a write operation directly to the Cache, Cache itself immediately writes the data to DB, Cache then confirms write success to the App

Caching Strategies in Distributed Caching - 2

Write-Back (Write-Behind) Strategy

Performs a write operation directly to the Cache, Cache confirms write success to the App, Later, asynchronously, the Cache itself writes the data to the DB

What is the Cache-Aside Pattern?

Application code is **responsible** for **managing cache interaction**

The **cache sits 'on the side'** of the **primary data store**

Read Operation Flow:

1. Application requests data from the Cache

2. Cache Hit: Data is returned from the cache

3. Cache Miss: Application reads data from the Primary Database, then stores it in the Cache, then returns data

Data is "lazy-loaded" into the cache on demand

Why Cache-Aside for Microservices?

Reduces Database Load
Significantly offloads read requests from the primary database for frequently accessed data

Improves Response Times (Latency)
Serving data from an in-memory distributed cache is mu
than DB reads

Increased Scalability
Services can handle more read requests without overwhelming their databases

Application Control
Developers have explicit control over what is cached and when

Drawbacks & Considerations for Cache-Aside

Data Staleness & Cache Invalidation
Cached data can become stale if the source data in the database changes. Requires a strategy to invalidate or update the cache

Cache Miss Penalty
The first request for uncached data (a miss) will incur the both the cache check AND the database fetch

"Thundering Herd" Problem
If a popular cached item expires, many concurrent requests might miss the cache simultaneously and all hit the database at once

Complexity in Application Code
The logic for checking the cache, fetching from DB, and populating the cache resides in the application

Choosing an appropriate TTL (Time-To-Live) / Eviction Policy

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture, EDA

Patterns &Principles

Database-per-Service

CAP Theorem, Database Sharding Pattern

Saga Pattern (Choreography/Orchestration)

Transactional Outbox Pattern, CDC

Cache Hit, Miss, Invalidation

Cache-Aside Pattern

Non-FR (Update)

High Scalability High Availability

Independent Deployability

Reduced Latency for Frequent Reads

Decreased Load on Primary Data Stores

Improved Application Responsiveness & Throughput

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ Central Distributed Cache

Microservices Distributed Caching apply Cache-Aside Pattern on API Gateway

1. User makes a request to an API gateway to retrieve some data
2. API gateway determines which microservice is responsible for handling the request and receives the request and checks to see if the data is in the cache
3. If the data is in the cache, API gateway retrieves it from the cache and returns it to the client
4. If the data is not in the cache, API gateway forward responsible microservice to retrieves it from the database, and then returns it to the API gateway
5. The API gateway receives the data from the microservice and stores it in the cache and returns it to the user

E-Shop's Development Technology Stack

Distributed Cache Store
Redis, Memcached

Cloud-Managed Cache Services
AWS ElastiCache (for Redis/Memcached), Azure Cache for Redis, Google Cloud Memorystore

redis

Amazon Elasticache

Evaluating Distributed Caching with Cache-Aside

Benefits

Significant Performance Boost

Reduced Load on Primary Databases

Improved Scalability & Throughput

Better User Experience

Shared Cache Consistency

Drawbacks

Cache Invalidation Complexity

Increased Network Hops

Cost & Management of Cache Cluster

Data Consistency between Cache and DB

Serialization Overhead

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Microservices Architecture

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

10-eshop-microservices-hybrid-cache

Way of Learning – The Course Flow

Where We Are: A Fleet of Services Ready for Production...

Our **E-Shop** now comprises **numerous independent microservices**

Each service can be **developed** and **tested** by its **own team**

Business demands rapid, **independent feature releases** for **each**

Need to **handle fluctuating traffic** with **varying scaling needs** per service

Questions

How do we manage the deployment, updates and scaling of all these services without constant manual effort, downtime, or chaos?

Problem: Deploy Microservices at Anytime w/ Zero-downtime and flexible scale

Problems

Manually deploying, updating and scaling a large number of services is error-prone, time-consuming

Considerations

Need for Zero-Downtime Deployments

Demand for Agile, Flexible Scaling

Complexity of Managing Many Moving Parts

Ensuring Consistent Environments

Solutions

Containers and Orchestrators

Deployment strategies; blue-green, rolling and canary deployment

Kubernetes Patterns; Sidecar Patterns, Service Mesh

DevOps and CI/CD Pipelines and IaC

Microservices Deployments with Containers and Orchestrators

Deployment strategies; blue-green, rolling and canary deployment

Kubernetes Patterns; Sidecar Patterns, Service Mesh Pattern

DevOps and CI/CD Pipelines

Infrastructure as code (IaC)

Mehmet Ozkaya

Architecture Design Journey

Architecture Design – Vertical Considerations

Problem: Deploy Microservices at Anytime w/ Zero-downtime and flexible scale

Problems

Manually deploying, updating and scaling a large number of services is error-prone, time-consuming

Considerations

Need for Zero-Downtime Deployments

Demand for Agile, Flexible Scaling

Complexity of Managing Many Moving Parts

Ensuring Consistent Environments

Solutions

Containers and Orchestrators

Deployment strategies; blue-green, rolling and canary deployment

Kubernetes Patterns; Sidecar Patterns, Service Mesh

DevOps and CI/CD Pipelines and IaC

Solution Path: Containers, Orchestrators, and Automation

Containers (Docker)

Orchestrators (Kubernetes)

Advanced Deployment Strategies

Kubernetes Patterns

DevOps & CI/CD Pipelines and Infrastructure as Code (IaC)

What are Containers?

A **standard**, **portable unit** of software that **packages up app code** and all **its dependencies** so it **can run quickly** and **reliably** from one computing environment to another

Solves the "**it works on my machine**" problem

Contains everything needed to run: code, runtime, system tools, libraries, and settings

Like a **physical shipping container** – standardizes transport regardless of the contents

How Do Containers Work?

Containers virtualize the **Operating System**, while **Virtual Machines (VMs) virtualize** the **Hardware**

VMs: Heavyweight. Each VM includes a full copy of a guest operating system. Start in minutes

Containers: Lightweight. Share the host machine's OS kernel. Use OS features like namespaces (for isolation) and cgroups (for resource limits). Start in seconds

https://www.docker.com/blog/containers-and-vms-together/

What are Container Runtimes and Registries?

Container Image
A lightweight, standalone, executable package – the blueprint for a container

Container Runtime
The low-level engine that actually runs containers (containerd, CRI-O). It manages the container lifecycle

Container Registry
A storage system and library for container images (Docker Hub, AWS ECR, Azure ACR). It's like "GitHub for Docker images

Containerization: Running Microservices in Containers

Containers and **microservices** are a **perfect match**

Each microservice runs in its **own isolated container environment**

Each container packages the **exact dependencies** its service needs, avoiding conflicts

A **service runs** the **same way everywhere**

Lightweight containers can be **started** and **stopped quickly**, **enabling rapid scaling**

What is Docker?

Docker is an **open platform** for **developing**, **shipping** and **running applications** inside **containers**

It **provides** the **tooling** to **make containerization easy** and **accessible**

It **simplifies** the **entire lifecycle** of an **application**, from **code** to **production**

Docker separates applications from the **underlying infrastructure**, enabling **rapid software delivery**

Docker: Images, Containers and Registries

Image
An immutable, lightweight, standalone template used to create containers. Built from a Dockerfile

Container
A running, live instance of an image. It's the encapsulated application process

Registry
A storage system for Docker images (Docker Hub)

Docker Architecture

Docker Client
The command-line interface (CLI) tool you use to interact with Docker (docker run, docker build)

Docker Daemon (Engine)
A background service (dockerd) that listens for API requests from the client and manages all Docker objects: images, containers, networks, and volumes

Docker Registry
The remote storage for images

https://www.docker.com/blog/containers-and-vms-together/

Application Containerization with Docker

1. **Write Dockerfile** for our application
2. **Build application** with this docker file and **creates** the **docker images**
3. **Run this images** on any machine and creates **running docker container from docker image**

Orchestrating whole **microservices** application with **Docker** and **Kubernetes**

What are Container Orchestrators?

Tools that **automate** the **deployment**, **scaling**, **networking** and **management** of **containerized apps** across a **cluster**

They **act as** the "**brains**" of a **containerized environment**

You **declaratively tell** the **orchestrator** your **desired state**

The **orchestrator works** to **make** the **actual state match** the **desired state**

Abstracts away the **complexity** of the **underlying individual servers**

Examples: Kubernetes (the industry standard), Docker Swarm, Apache Mesos

Why Use Orchestrators for Microservices Deployments?

How container instances be provisioned into cluster of multiple machines ?

After deployment, how will containers discover and communicate with each other ?

How can containers scale in or out on-demand and peek traffic ?

How do you monitor the health of each container ?

How do you protect containers against hardware and software failures ?

How do upgrade containers for a live application with zero downtime ?

Application Containerization and Orchestration

1. **Package** each of **microservices** into a **container image** and **push** it to a **container registry**
2. **Then** use the **container orchestrator** to **deploy** the **microservices** to a cluster of nodes
3. **Orchestrator** will handle tasks such as **scheduling containers** onto nodes, **monitoring** the health of the containers, and providing **self-healing** capabilities if a container fails

What is Kubernetes?

Kubernetes (**k8s**) is an **open-source container orchestration platform** for **automating** the **deployment**, **scaling** and **management** of **containerized apps**

Originally **developed by Google**, now **maintained** by the **Cloud Native Computing Foundation** (**CNCF**)

It's a **portable**, **extensible platform** for **managing containerized workloads** and **services**

Facilitates declarative configuration and **automation**

I **want 3 instances** of **my Catalog Service** running with this **container image**

Designed to be **platform-agnostic**

kubernetes

Why Kubernetes for Microservices

Self-healing
Automatically restarts failed containers and reschedules them on healthy nodes

Automatic Scaling
Horizontally scales service instances up or down based on resource usage (CPU/memory) or custom metrics

Service Discovery & Load Balancing
Provides stable DNS names for services and distributes network traffic to available instances

Automated Rollouts & Rollbacks
Manages updates to your application in a controlled way, allowing for easy rollbacks if something goes wrong

Kubernetes Architecture: Control Plane & Worker Nodes

https://kubernetes.io/docs/concepts/overview/components/

Kubernetes Components

Pods
The smallest deployable unit. A group of one or more containers that share network and storage

ReplicaSet
Ensures a specified number of replica Pods are always running

Deployments
Manages ReplicaSets and provides declarative updates (for rolling out new versions)

Service
Exposes a set of Pods as a network service with a stable DNS name and IP address, and provides load balancing

ConfigMap & Secret
For managing app configuration and sensitive data (like passwords, API keys) externally from the container image

Volume: For providing persistent storage to containers

Simplifying Deployments with Helm Charts

Helm is the package manager for Kubernetes

Bundles all necessary K8s configuration files (for Deployments, Services, ConfigMaps..) into a single, reusable package called a Chart

Simplifies complex application deployments
Instead of manually applying many YAML files, you install one chart

Enables reusability and sharing
Easily deploy the same application across different environments (dev, staging, prod) with different configurations

Manages application releases
Supports easy versioning, upgrades, and rollbacks

The Sidecar Pattern

A **design pattern** where a **helper container** (**Sidecar**) **runs alongside** the **main application container** within the **same Pod**

The **sidecar** performs **auxiliary** or **cross-cutting tasks** (**logging**, **monitoring**, **configuration**, **proxying**)

The **sidecar enhances** the **main application without being part** of its **codebase**

How Does the Sidecar Pattern Work & Why Use It?

Decoupling & Separation of Concerns
Moves cross-cutting logic out of app code, keep microservices focused on its business purpose

Reusability
The same sidecar container can be attached to many different microservices without any code changes in those services

Language Independence
It can be written in different languages than the main app

Independent Updates
You can update the functionality of the sidecar without having to rebuild or redeploy your main app container

The Service Mesh Pattern

A **dedicated infrastructure layer** for **managing service-to-service communication**. It's a **system-wide** application of the **Sidecar pattern** for **networking**

Data Plane
A network of Sidecar Proxies (Envoy) deployed alongside every microservice instance. All service traffic is intercepted and routed by these proxies

Control Plane
The central "brain" that configures and manages all the sidecar proxies in the data plane

How Does the Service Mesh Pattern Work & Why Use It?

The **service mesh offloads complex networking** logic from your services into the **sidecar proxies**

Traffic Management
Advanced routing (canary releases, A/B testing), load balancing, traffic splitting

Resilience
Automatic retries, timeouts, and circuit breaking for all inter-service calls, without any application code changes

Security
Enforces encrypted communication between services (mutual TLS - mTLS) and access control policies

Observability
Automatically generates detailed metrics, logs, and traces for all network traffic between services

What is DevOps?

DevOps is a **cultural philosophy** and **set of practices** that emphasizes **collaboration** between **Development** and **Operations** teams

Break down silos to **improve** the **speed** and **quality** of **software delivery**

Microservices & **DevOps** is a **perfect match**. The **"You build it, you run it"** principle, where a **small team owns** a **microservice** from **development through** to **production**, is the **embodiment** of **DevOps culture**

What is CI/CD?

CI (Continuous Integration): Developers frequently merge code into a central repository, automatically triggering builds and tests to catch bugs early

CD (Continuous Delivery/Deployment): Automatically release code that passes CI

- **Delivery:** Ready for manual push-button deployment
- **Deployment:** Automatically deployed to production

Automates the **complex process** of **building, testing** and **deploying dozens or hundreds** of **independent services**

CI/CD Pipeline Steps for Microservices Deployments

CI/CD Flow for Microservices Deployments

CI/CD Tools for Microservices Deployments

SCC

- Github
- Gitlab
- AWS CodeBuild

Build

- Jenkins
- Maven
- Gradle

Registry

- DockerHub
- ACR
- GCR

Deployment

- Helm Charts
- Ansible Scripts
- Azure Devops
- AWS CodeBuild

Monitor

- Prometheus
- Datadog
- Amazon CloudWatch
- Azure Monitor

IaC

- Terraform
- Ansible
- AWS CDK

Rolling Deployment

Gradually replace instances of the **old version** with **instances** of the **new version**, **one** or a **few at a time**

The **orchestrator** (e.g. **Kubernetes**) ensures a **minimum number** of **instances** are **always running**

Pros: Simple (often the default in orchestrators), zero-downtime, low resource overhead

Cons: Slow for large deployments, temporary state of mixed versions (v1 & v2 run simultaneously), rollback is a slow reverse process

Blue-Green Deployment

Deploy a **new version alongside** the **old version** in a **separate**, **identical environment**, then **switch traffic instantly**

Blue Environment: The current live production version (v1)

Green Environment: The new version (v2), deployed and tested in parallel

Pros: Instantaneous cut-over and rollback (just flip the router), new version is fully tested in a production-like setting before receiving live traffic

Cons: Costly and resource-intensive (requires double the infrastructure), managing stateful applications and database changes can be complex

Canary Deployment

Gradually release the **new version** to a **small subset** of **users/requests first**, **before rolling** it out to **everyone**

The **small initial group** acts as a "**canary in a coal mine**" to **detect issues**

Pros: Lowest risk deployment, allows for real-world production testing with live traffic, minimizes impact of a bad release

Cons: Most complex to implement (requires sophisticated traffic shaping), heavy reliance on excellent monitoring and observability

What is Infrastructure as Code (IaC) ?

A **practice** of **managing** and **provisioning infrastructure through code** and **configuration files**, **rather than** through **manual processes**

Treat your **infrastructure like application code**: **store** it in **version control**, **review** it, and **automate** its **deployment**

Goal: Make infrastructure provisioning automated, repeatable, and consistent

IaC Usage & Benefits in Cloud-Native Microservices

Rapid Provisioning of Infrastructure

Consistency Across Environments

Enables Immutable Infrastructure

Version Control and Rollbacks

Integration with CI/CD Pipelines

Explore: Cloud-Native IaC Tools

Cloud-Native IaC tools

- Terraform
- Ansible
- Puppet
- Chef
- Pulumi

Cloud Serverless IaC tools

- AWS CloudFormation
- AWS SAM (Serverless Application Model)
- AWS Cloud Development Kit (CDK)
- Azure Resource Manager (ARM) Templates
- Azure Bicep
- Google Cloud Deployment Manager

HashiCorp Terraform

AWS CloudFormation

Terraform IaC Steps - How Terraform Works ?

https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code

What is GitOps?

GitOps is an **operational framework** that **uses** a **Git repository** as the **single source of truth** for **both infrastructure** and **application configurations**

It **applies DevOps best practices** (**version control**, **collaboration**, **CI/CD**) directly to **infrastructure** and **deployment** management

It's an **evolution** of **Infrastructure as Code** (IaC)

The **desired state** of your entire system is **described declaratively** in **Git**.

An **automated process** ensures the **live production environment** always **matches** the **state** of the **Git repositor**

Key Principles of GitOps

Declarative Configuration
The entire system state is described declaratively in files (e.g., Kubernetes YAML)

Version Control as the Single Source of Truth
Git holds the desired state. If it's in Git, it's the truth

Automated Synchronization (Pull-Based)
An automated agent inside the cluster continuously pulls changes from Git and applies them

Continuous Reconciliation
The agent constantly compares the live state with the state in Git and corrects any drift

The Steps of GitOps Workflow

GitOps works by using **Git** as the **single source of truth** for declarative infrastructure. Its an operational framework that takes DevOps, CI/CD and IaC best practices. **4 main steps:**

1. **Define the Desired State in Git**
2. **Submit Change through Pull Request**
3. **Automated Synchronization**
4. **Continuous Feedback and Monitoring**

GitOps Workflow Architecture

GitOps Real-world Use Case - Black Friday Sale EShop

1- Define the Desired State in Git
2- Submit Change through Pull Request
3- Automated Synchronization
4- Continuous Feedback and Monitoring
5- Rollbacks and Audit Trails

Explore: Cloud-Native GitOps Tools

Cloud-native GitOps tools are designed to facilitate the GitOps workflow, particularly in Kubernetes ensuring that the state of your cluster matches the configuration defined in a Git repository.

Cloud-Native GitOps tools

- ArgoCD
- Flux
- Jenkins X
- Codefresh
- Tekton

Cloud Serverless GitOps tools

- AWS Proton
- Azure Arc
- Google Config Sync

Argo CD deploy microservices in Kubernetes with CI/CD

Step 1: Define Application Code and Infra Code in Git
Step 2: Integrate CICD Pipelines with GitOps
Step 3: Submit Change in Application Code
Step 4: CICD Update Docker Image Tag into Infra Code
Step 5: GitOps Automated Synchronization
Step 6: Continuous Monitoring and Audit Trails

https://www.gspann.com/resources/blogs/continuous-delivery-for-kubernetes-with-gitops-and-argo-cd/

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture, EDA

Patterns &Principles

Database-per-Service

CAP Theorem, Database Sharding Pattern

Saga Pattern (Choreography/Orchestration)

Transactional Outbox Pattern, CDC

Helm Charts

Sidecar Pattern, Service Mesh Pattern

Non-FR (Update)

High Scalability High Availability

Independent Deployability

Containers, Orchestrators

DevOps Culture, CI/CD Pipelines, Infrastructure as Code (IaC), GitOps

Blue-Green, Rolling, Canary Deployments

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Design EShop Microservice w/ DevOps, CI/CD, IaC, GitOps

Containerization
Orchestration
Packaging (Helm)
Service Mesh (Sidecar)
Automation (DevOps & CI/CD)
Deployment Strategy
Infrastructure as Code (IaC)
GitOps

Tools: Cloud-Native DevOps, CI/CD, IaC and GitOps

Containers
- Docker

Orchestrator
- Amazon EKS

Container Registry
- Amazon ECR

CI/CD Pipeline
- GitHub Actions

IaC
- Terraform

GitOps
- ArgoCD

Steps of DevOps,CI/CD, IaC and GitOps Architecture

1. **Infrastructure Provisioning:** Terraform configuration files to describe your AWS infrastructure including the EKS cluster, ECR repositories.
2. **Set Up ArgoCD:** Deploy ArgoCD in your EKS cluster. Configure ArgoCD to monitor your GitHub repositories for changes in Kubernetes manifests.
3. **Develop Microservices:** Writes code for the microservices and pushes to GitHub.
4. **GitHub Actions Trigger:** It builds a Docker image of the microservice. Pushes the Docker image to ECR. Updates the Kubernetes manifest files with the new image tag.
5. **ArgoCD Observes and Deploys:** ArgoCD notices the changes in Kubernetes manifests. It pulls these manifest files. Deploys the updated microservice to the EKS cluster by applying the manifests.
6. **Monitor and Observe:** Set up monitoring and logging solutions like Prometheus and Grafana, AWS CloudWatch for observing the performance and health of your microservices.

Microservices w/ Containers and Orchestrators

Frontend SPAs
- Angular
- Vue
- React

API Gateways
- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Backend Microservices
- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS

CI/CD
- GitHub Actions

GitOps
- ArgoCD

Container Tools
- Docker
- Containerd
- Podman

Orchestrator Tools
- Kubernetes
- Helm Charts

Cloud Orchestrator
- AKS, EKS, GKS
- AWS Elastic Beanstalk
- Azure Container Apps

Service Mesh
- Istio
- Linkerd

IaC
- Terraform
- Ansible

Cloud IaC
- AWS CDK

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Microservices Architecture

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

10-eshop-microservices-hybrid-cache

Way of Learning – The Course Flow

Where We Are: Services Ready for Production

Our E-Shop containers run consistently everywhere

Kubernetes can automatically scale services

CI/CD and GitOps provide automation

Kubernetes can restart failed containers

Questions

How do we handle failures in distrubuted system within all deployed services ?

Problem: Fault tolerance Microservices able to remains operational for any failures or disruptions

Problems

Business teams able to deploy new features immediately to compete the market

Due to heavy traffic, microservices got exceptions and broke some major use cases that cause lost revenue

Create big crisis to customer loyalty

Considerations

Deploy new features immediately without affecting the system general behaviors

System should recover from failures and provide ability of a syste to withstands

Architecture should be designed to be resilient

Solutions

Microservices Resilience and Fault Tolerance

Microservices Observability with Distributed Logging and Distributed Tracing

Microservices Health Monitoring

Microservices Resilience, Observability and Monitoring

Microservices Resilience and Fault Tolerance

Retry, Circuit-Breaker, Bulkhead Pattern

Microservices Observability with Distributed Logging and Tracing

Elastic Stack which includes Elasticsearch + Logstash + Kibana

Microservices Health Monitoring

Mehmet Ozkaya

Architecture Design Journey

Architecture Design – Vertical Considerations

Problem: Fault tolerance Microservices able to remains operational for any failures or disruptions

Problems

Business teams able to deploy new features immediately to compete the market

Due to heavy traffic, microservices got exceptions and broke some major use cases that cause lost revenue

Create big crisis to customer loyalty

Considerations

Deploy new features immediately without affecting the system general behaviors

System should recover from failures and provide ability of a syste to withstands

Architecture should be designed to be resilient

Solutions

Microservices Resilience and Fault Tolerance

Microservices Observability with Distributed Logging and Distributed Tracing

Microservices Health Monitoring

The Three Pillars of a Robust System

Resilience

Observability

Monitoring

What We'll Learn in This Section: Our Roadmap

Resilience & Fault Tolerance Patterns
Retry, Circuit Breaker, Bulkhead Pattern

Observability with Distributed Tracing & Logging
Elastic Stack (Elasticsearch, Logstash, Kibana) for centralized logging, OpenTelemetry and tools like Zipkin or Jaeger for distributed tracing

Monitoring & Health Checks
Microservice Health Checks (Liveness & Readiness Probes for Kubernetes), Monitoring with Prometheus & Grafana

The Challenge: Cascading Failures in Microservices

In a distributed system, services communicate over a network, which introduces potential points of failure

A service might be slow or temporarily unavailable

Network issues can cause requests to fail

A failure in one downstream service can block and crash upstream services that depend on it

What is Microservices Resilience?

Ability of a system to withstand and recover from failures while remaining operational from the user's perspective

It's not about preventing all failures (which is impossible)

It's about designing for failure and accepting partial failures

App can return to a fully functioning state or at minimum, degrade functionality gracefully without a complete outage

Microservices Resiliency Patterns

Retry Pattern

Circuit Breaker Pattern

Bulkhead Pattern

Timeout Pattern

Fallback Pattern

The Retry Pattern: Handling Transient Failures

A pattern that enables an application to automatically retry a failed operation with the expectation that the failure is transient (temporary)

Problem

Many failures in distributed systems are short-lived (momentary server overload, temporary database locks)

Solution

Instead of immediately failing, retry the operation e

The Limitation of Retry & The Need for Circuit Breaker

What happens if a failure is not transient?

If a service is completely down or has a persistent bug, retrying repeatedly is harmful

It wastes resources (CPU, threads, network) in the calling service

It can overwhelm the failing service, preventing it from recovering (a "denial of service" attack on yourself)

It leads to long timeouts and poor user experience

The Circuit Breaker Pattern

A pattern that acts as a stateful proxy to prevent an application from repeatedly trying to execute an operation that is likely to fail

Protects the system from cascading failures

Allows a failing service time to recover

Circuit Breaker States

Retry and Circuit Breaker for Robust Resilience

Retry Pattern
Handles intermittent, transient failures

Circuit Breaker Pattern
Handles systemic, longer-lasting failures

Combined Flow
A request first passes the Circuit Breaker. If the circuit is Closed, the Retry logic is attempted

If repeated retries cause the failure threshold to be met, the Circuit Breaker opens, and subsequent requests will fail immediately before even attempting a retry

The Bulkhead Pattern

Isolates resources used by different downstream service calls or workloads within a single service

Based on the bulkheads in a ship that prevent a single leak from sinking the entire vessel

Goal: Prevent a failure or resource exhaustion in one part of a system from cascading and bringing down the entire service

API Gateway's thread pool for Catalog Error

The Timeout Pattern

Problem: A downstream service may become slow or hang, causing the calling service's resources (threads) to be blocked indefinitely

Solution: Don't wait forever. If a response isn't received within the configured timeout, abandon the request and throw a timeout exception

The Fallback Pattern

Provides an alternative course of action or response when a primary operation fails

Enables graceful degradation instead of complete failure

Serving data from a cache, Returning a default value or a simplified response

What is Microservices Observability?

The ability to understand the internal state of a complex system by asking questions from the outside, based on the data it generates

It's more than just monitoring; it's about debuggability in production

In a monolith, you could use a debugger or look at a single log file. In microservices, this is impossible

Goal: To understand why something is failing or performing poorly in a distributed environment

Key Pillars: Distributed Logging, Distributed Tracing, and Metrics

Distributed Logging: Centralizing Your Story

The practice of collecting, aggregating, and analyzing log data from all your distributed microservice instances in a single, centralized location

The problem of logs being scattered across dozens or hundreds of servers/containers

Searching, filtering, and correlating logs from across the entire system to diagnose issues

Popular Toolset: The Elastic Stack (ELK

Real-world Example of Microservices Observability with Distributed Logging and Distributed Tracing

Implement **distributed logging** to **collect**, **store**, and **analyze log data** from service instances, include information about **service calls**, **errors**, **performance metrics**, and other **relevant data**

Implement **distributed tracing** to **track the flow of requests** through the system

If a user **adds an item to their shopping cart**, the **trace with information** about the request flows from the **shopping cart** service to the **payment** processing service to the **product catalog** service

Elastic Stack for Microservices Observability with Distributed Logging

Elastic Stack is a collection of **open-source tools** for **collecting**, **storing**, and **analyzing log data** and other types of data.

Used for **microservices observability** to **provides** a **flexible** and **scalable** platform for **monitoring** and understanding the behavior.

Elasticsearch
Distributed search and analytics engine that can be used to store and index log data and other types of data.

Logstash
Data collection and transformation tool that used to collect log data from different sources and send it to Elasticsearch.

Kibana
Data visualization and exploration tool that used to create dashboards and visualizations based on data.

Beats
Collection of lightweight data shippers that used to collect log data and send it to Elasticsearch.

Why logging with ElasticSearch and Kibana in Microservices ?

Logging is **critical** to **implement** to **identify problems** in distributed architecture.

Elastic Stack used to **collect**, **store**, and **analyze log** data from multiple service instances in a microservices architecture.

Useful for **understanding** the **behavior** of the system over time, **identifying patterns** and **trends**, and **troubleshooting issues**.

Elastic Stack might be used to **collect log** data from each service instance.

Kibana to **visualize** and **analyze** the data in order to **identify patterns** or **trends**.

Real-world Example with ElasticStack

Distributed Tracing: Following the Request Journey

The practice of tracking the flow of a single request as it travels through multiple microservices in a distributed system

The problem of understanding latency and identifying bottlenecks. Answers "Why was this request slow?

Key Standard: OpenTelemetry (OTel): A vendor-neutral open-source standard for generating and collecting telemetry data (traces, metrics, logs)

Key Tools (Backends): Zipkin, Jaeger

Distributed Tracing with OpenTelemetry using Zipkin

Distributed Tracing is used to **track the flow of a request** as it is processed by different microservices in a system.

Different microservices are interacting and **identify issues** or **bottlenecks** and **troubleshooting issues** in the system.

OpenTelemetry is an **open-source project** that provides a set of APIs for **collecting** and **exporting telemetry data**; **traces**, **metrics**, and **logs**.

Zipkin is a **distributed tracing system** that **collects** and **stores trace** data from microservices, provides a web UI for **viewing** and **analyzing trace data**.

To use **OpenTelemetry with Zipkin** for **microservices distributed tracing**, **OpenTelemetry SDK** would be **integrated.**

Allows to collect trace data as requests flow through the system, and **send data** to a **Zipkin server** for **storage** and **visualization**.

Real-world Example of Distributed Tracing with OpenTelemetry using Zipkin

Use **OpenTelemetry** with **Zipkin** to **collect trace** data **as requests flow** through the system.

When a **user adds** an item to their **shopping cart**, the **trace include information about** the **request**.

The **trace data collected by** the **OpenTelemetry SDKs** integrated into each service instance, and then **sent** to a **Zipkin server** for **storage** and **visualization**.

Use the **Zipkin UI** to **visualize** and **analyze** the **trace data**, to **identify patterns** or **trends**.

What are Health Checks?

a specific endpoint exposed by a microservice (e.g., /health) that returns its current operational status

It's a simple way to answer the question: Are you okay and ready to work?

Crucial for automation: Enables orchestrators to make automated decisions about restarting services or routing traffic

Main Types: Liveness (is it running?) and Readiness (is it ready for traffic?)

Microservices Health Checks: Liveness, Readiness and Performance Checks

- The **process of monitoring** the **health** and **performance** of **individual microservices** in a system.
- The **failure of a single microservice** can have **cascading effects** on the rest of the system, **important to identify** and **address issues**.
- What is **Health Checks** for **microservices** ?
- **Health Checks for microservices** are a way to **monitor the health** and **performance** of **individual microservices** in a system.
- **Health checks** used to **determine whether** a **microservice** is **functioning properly** and is **able to handle requests**.
- There are **3 types of health checks** that can be used for microservices.

Microservices Health Checks: Liveness, Readiness and Performance Checks

- **Liveness Checks**
  Determine whether a microservice is still running. If a liveness check fails, it may indicate that the microservice has crashed.
- **Readiness Checks**
  Determine whether a microservice is ready to handle requests. If a readiness check fails, it may indicate that the microservice is not yet ready to handle traffic.
- **Performance Checks**
  Monitor the performance of a microservice, such as response times or error rates. If the results of a performance check indicate that a microservice is not performing as expected.

Health Checks in Kubernetes: Liveness & Readiness Probes

Kubernetes uses Probes to implement health checks, enabling self-healing and safe deployments

Liveness Probe

If this probe fails, Kubernetes restarts the container

Readiness Probe

If this probe fails, Kubernetes stops sending traffic to the pod until the probe succeeds again

Monitoring with Prometheus & Grafana

While health checks are binary (up/down), Monitoring tracks performance and behavior over time by collecting metrics

Prometheus
An open-source monitoring and alerting system. It periodically pulls (scrapes) time-series metrics from endpoints exposed by your services

Grafana
An open-source visualization platform. It connects to data sources like Prometheus to create powerful, interactive dashboards for visualizing your metrics

Prometheus Architecture

https://prometheus.io/docs/introduction/overview/

How Prometheus Works ?

How Prometheus Works ?

Setup and Configuration of Prometheus in Kubernetes

- Deploy Prometheus to your Kubernetes cluster.
- Configure Prometheus to discover and scrape metrics from your microservices.

Instrumenting Microservices

- Implement metric exposition in your microservices using Prometheus client libraries.
- Expose metrics through an HTTP endpoint (usually /metrics).

Service Discovery

- Use Prometheus's Kubernetes service discovery to dynamically discover services to monitor.

Defining Recording Rules and Alerts

- Use recording rules to precompute frequently needed or computationally expensive expressions.

Querying Metrics with PromQL

- Use PromQL, Prometheus's flexible query language, to query your metrics.

Completing the Picture: Tying it All Together

Resilience, Observability, and Monitoring work together to create a robust production system

Way of Learning – The Course Flow

Before Design – What we have in our Design Toolbox ?

Architectures

Monolithic Architecture

Modular Monolith Architecture

Microservices Architecture, EDA

Patterns &Principles

Database-per-Service

CAP Theorem, Database Sharding Pattern

Saga Pattern, Transactional Outbox Pattern, CDC

Resilience & Fault Tolerance Patterns: Retry, Circuit Breaker, Bulkhead, Timeout, Fallback

Non-FR (Update)

High Scalability High Availability

Independent Deployability

DevOps Culture, CI/CD Pipelines, Infrastructure as Code (IaC), GitOps

Blue-Green, Rolling, Canary Deployments

Distributed Logging, Distributed Tracing, Monitoring & Health Checks

FR

List products

View product details

Add products to a shopping basket

Checkout the shopping cart and place an order

View past orders

Before Design – What we have in our design toolbox ? - New

Architectures

- **Microservices Architecture**
- Event-Driven Microservices Architecture

Patterns&Principles

- The Database-per-Service Pattern, Polygot Persistence, Decompose services by scalability, The Scale Cube
- Microservices Decomposition Pattern
- Microservices Communications Patterns
- Microservices Data Management Patterns
- Event-Driven Architecture
- Microservices Distributed Caching
- Microservices Deployments with Containers and Orchestrators

Microservices Resilience

- **Resilience Patterns; Retry, Circuit-Breaker, Bulkhead, Timeout, Fallback Pattern**
- **Distributed Logging and Distributed Tracing**
- **Elastic Stack; Elasticsearch + Logstash + Kibana**
- **OpenTelemetry using Zipkin**
- **Kubernetes Health Monitoring with tools like Prometheus and Grafana**

Non-FR

- High Scalability
- High Availability
- Millions of Concurrent User
- Independent Deployable
- Technology agnostic
- Data isolation
- Resilience and Fault isolation

FR

- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts
- Checkout the shopping cart and create an order
- List my old orders and order items history

- **Microservices Resilience, Observability and Monitoring**

Design EShop Microservice w/ Resilience, Observability and Monitoring

Microservices Resilience, Observability and Monitoring

- **Improved fault tolerance**
  Help a microservices system continue functioning even in the face of failures or other disruptions, making it more fault tolerant.
- **Improved uptime and availability**
  Implementing patterns circuit breakers and fallback mechanisms, you can ensure that your system remains available even when individual components get problems.
- **Increased reliability**
  When a microservices system is designed to be resilient, it is less likely to experience failures or disruptions.
- **Enhanced scalability**
  Microservices architecture is designed to be scalable, and using resilience patterns can further enhance this capability.
- The bulkhead pattern can allow you to scale different components of your system independently, without affecting the others.

Adapt: Microservices Resilience, Observability and Monitoring

Frontend SPAs
- Angular
- Vue
- React

API Gateways
- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Backend Microservices
- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS

Resiliency Pattern
- Retry
- Circuit breaker
- Bulkhead
- Timeout, Fallback

Distributed Logging
- Elastic Stack
- Fluentd
- Splunk

Cloud Distributed Logging
- AWS Elastic Cloud
- Azure Elastic

Distributed Tracing
- OpenTelemetry
- Zipkin
- AWS X-Ray

Monitoring
- Kubernetes Health Checks
- Prometheus – Grafana, Datadog

Cloud Monitoring
- Azure Application Insights
- Amazon CloudWatch

Way of Learning – The Course Flow

Our "Code Exploration" Approach

Examine Codes
AppHost + Projects and codes, put break points

Run App
See Aspire Dashboard and graph -projects env variables

Test
E2E use cases see products, add basket, checkout, see orders

Observe
Logs, Traces, Db records, caches, buses

Code: E-Shop Microservices Architecture

Language: C#

Framework: .NET / ASP.NET Core

IDE/Tooling: Visual Studio

Path: EShopDesign_Sections

10-eshop-microservices-hybrid-cache

Serverless Microservices Architecture

AWS Event-driven Serverless Microservices using AWS Lambda, API Gateway, EventBridge, SQS, DynamoDB and CDK for IaC

Mehmet Ozkaya

Architecture Design Journey

Introduction - Serverless Microservices

- **Serverless microservices** in which the **infrastructure** is managed by a **cloud provider** and the microservices are **executed** in a **serverless computing** environment.
- **Microservices** are **automatically scaled** to meet the **demand of the application** and developer **does not worry about provisioning** and **managing servers**.
- **AWS Lambda**, **Azure Functions** and **Google Cloud Functions** are a **serverless computing** platforms provided by AWS, Microsoft Azure and Google Cloud.
- It allows developers to **write** and **deploy microservices** that **run in response to specific events**, such as an **HTTP request** or a **change to a database**.

Example of Serverless Microservices

- **Write the microservice code** in a programming language supported by AWS Lambda, such as Node.js, Python, or Java.
- **Package the code** and any dependencies into a deployable package, such as a ZIP file.
- **Create an AWS Lambda function** and upload the package.
- **Configure the function** to be triggered by a specific event, such as an HTTP request.
- **Test** and **debug the function** using the AWS Lambda console or command-line tools.
- **Deploy the function** to a production environment and monitor its performance using AWS CloudWatch.

AWS Serverless services used

Compute

AWS Lambda

Databases

Amazon DynamoDB

Messaging/Application Integrations

Amazon EventBridge

Amazon Simple Queue Service (Amazon SQS)

API Management

Amazon API Gateway

IaC

AWS CloudFormation

AWS CDK Stack

Monitoring

Amazon CloudWatch

AWS Lambda as a Microservice

- Microservice are **small business services** that can work together and can be deployed **autonomously / independently**.
- **Lambda** is a service that allows you to run your functions in the cloud **completely serverless** and eliminates the operational **complexity**.
- It **integrates** with the **API gateway**, allows you to invoke functions with the API calls, and makes your architecture completely serverless.
- Microservice with **AWS Lambda**, which removing the architectural overhead of designing for **scaling** and high **availability**,
- Eliminating the **operational efforts** of operating and monitoring the microservice's underlying infrastructure.

https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/serverless-microservices.html

Serverless E-Commerce Microservice Architecture

- Developed in Serverless environment
  - developing with AWS Lambda
  - No Deployment Requirement
  - and IaC by AWS CDK CloudFormation
- **AWS API Gateway**
- **Microservices** developed with AWS Lambda
- **Databases** are no-sql DynamoDB that can be key-value pair or document databases
- **Message broker system** which is Amazon EventBridge Eventbus
- **Amazon CloudWatch** for centralized logging and monitoring
- By default automatic scalability and high avaliability

Design: Serverless E-Commerce Microservices

Thanks

Thank you so much for being with me on this journey.

Reviews and feedback is really encourage to me for pushing forward to create new courses like this.

Mehmet Ozkaya