

Java Microservices

Mert Alptekin

Lead Software Consultant

Eğitim Kataloğu

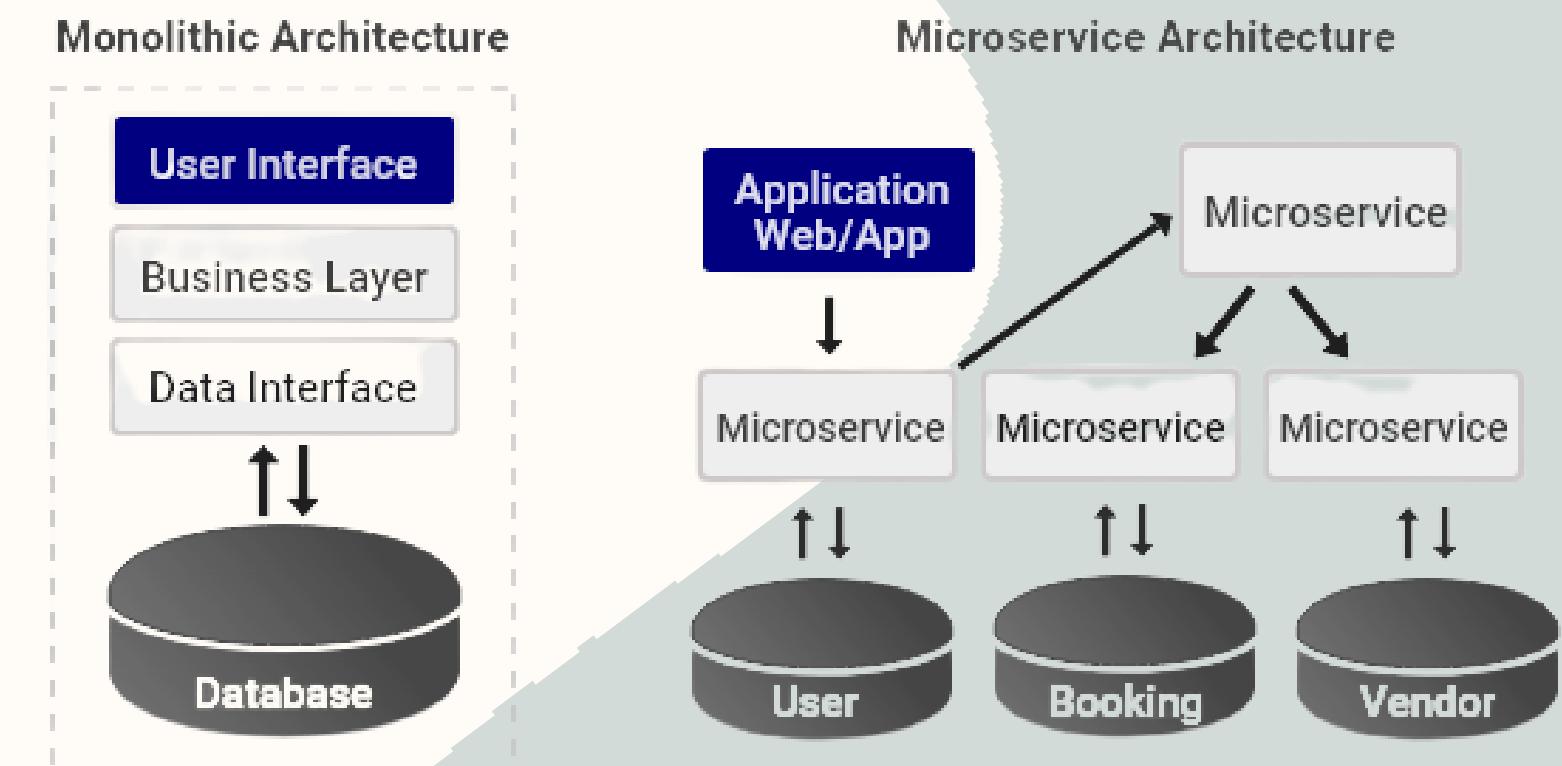
- 1 Mikro Hizmet Mimarilerine Giriş
- 2 Mikro Hizmetler Arası Sınırların Belirlenmesi
- 3 Spring Cloud
- 4 Spring Cloud Config
- 5 Spring Cloud Admin
- 6 Spring Cloud Netflix Eureka
- 7 Mikro Hizmet Mimarilerde İletişim
- 8 Mikro Hizmet Mimarilerde Resiliency
- 9 Spring Cloud Gateway, Spring Cloud Loadbalancer
- 10 Keycloak Identity Server
- 11 Mikro hizmet Mimarilerde Veritabanı tasarıımı
- 12 Spring Cloud Stream (Kafka)
- 13 Distributed Transaction
- 14 Distributed Tracing & Logging

Mikro Hizmet Mimarileri

Mikro hizmet mimarisi, yazılım sisteminin birbirinden bağımsız, küçük, dağıtık servisler halinde geliştirildiği, konuşlandırıldığı ve çalıştırıldığı bir mimari yaklaşımıdır.

Her mikro servis:

- Tek bir iş sorumluluğu taşır.
- Kendi veritabanına sahip olabilir.
- Kendi başına deploy edilebilir.
- Diğer servislerle genellikle **HTTP**, **gRPC**, **AMQP** gibi protokoller üzerinden haberleşir



Hangi Proje Tiplerine Uygundur?

- Büyük ölçekli, karmaşık uygulamalar
- Sürekli büyüyen, hızla gelişen projeler
- Çoklu ekiplerin paralel çalıştığı projeler
- Bulut tabanlı SaaS ve dağıtık sistemler
- Farklı tipte veri kaynağı ihtiyacı olan sistemler
- Farklı servisler, farklı iş yüklerine göre optimize edilebilir (örneğin batch işlerini ayrı, real-time işlemleri ayrı servislerde tutmak)
- Global ölçekte (coğrafi olarak dağıtık) çalışan uygulamalar

Uygulanması Zor Olan Proje Tipleri

- **Çok katı regülasyonların olduğu sektörler**

- (Bankacılık, sağlık, devlet uygulamaları)
- Merkezi denetim ve veri bütünlüğü zor sağlanabilir

- **Küçük ve basit uygulamalar**

- Mikro servislerin getirdiği operasyonel yük ve karmaşıklık gereksiz olur
- Tek bir ekip tarafından yönetilen ve küçük ölçekli projelerde monolith tercih edilir

- **Miras (legacy) sistemler**

- Mikro servislere dönüştürmek çok büyük refactor gerektirir
- Kademeli geçiş planları gereklidir

- **Aşırı karmaşık domainlerin çok sık değiştiği durumlar**

- Servis sınırlarının belirlenmesi (Bounded Context) zorlaşır
- Domain odaklı karmaşık bağımlılıklar yönetimi zorlaşır

Mikro Hizmet Mimarilerin Avantajları

- **Bağımsız geliştirme ve dağıtım**
 - Servis bazlı CI/CD pipeline'ları ile bağımsız deploy ve rollback imkanı
- **Küçük ve odaklanmış servisler**
 - Tek sorumluluk prensibi (Single Responsibility Principle) ile yüksek cohesion ve düşük coupling
- **Farklı teknolojilerin birlikte kullanımı**
 - Polyglot programming ve heterojen teknoloji yığını desteği
- **Daha iyi hata izolasyonu**
 - Fault isolation sayesinde bir servisin hatasının sistem genelini etkilememesi
- **Kolay ölçeklenebilirlik**
 - Servis bazlı yatay ölçeklendirme (**scale-out**) ile kaynak optimizasyonu

Mikro Hizmet Mimarilerin Dezavantajları

- **Dağıtık sistem karmaşıklığı**
 - (network partition, latency, fault tolerance sorunları)
- **Servisler arası iletişim problemleri**
 - (RPC çağrılarında gecikme, hata yönetimi)
- **Yüksek operasyonel maliyet**
 - (Devops süreçlerinin karmaşıklığı)
- **Güvenlik, versiyonlama ve API sözleşme yönetimi zorluluğu**
 - (backward compatibility, token yönetimi)
- **Dağıtık veri tutarsızlığı ve senkronizasyon sorunları**
 - (CAP teoremi, eventual consistency, distributed transactions yönetimi zorluğu)
- **Servislerin izlenmesi ve hataların tespiti için kapsamlı gözlemlenebilirlik gereksinimi**
 - (distributed tracing, log korrelasyonu, merkezi monitoring altyapısı gereksinimi)

Mikro Hizmet Mimarilerde Dikkat Edilmesi Gerekenler

• Servis Sınırlarının Doğru Belirlenmesi

- Hizmetler mümkün olduğunca bağımsız ve gevşek bağlı olmalı.
- Domainde anlamlı ve tutarlı sınırlar çizilmeli

• API Tasarımı

- Mikro hizmetler arasındaki iletişim için net, tutarlı ve iyi dokümantel edilmiş API'ler oluşturulmalı.
- REST, gRPC, GraphQL gibi uygun iletişim protokollerini seçilmeli.
- API versiyonlama ve geriye dönük uyumluluk stratejileri belirlenmeli

• Bağımsız Dağıtım ve Ölçeklenebilirlik

- Continuous Integration/Continuous Deployment (CI/CD) süreçleri otomatik ve sağlam olmalı.
- Her mikro hizmet bağımsız olarak deploy edilebilmeli.
- Servisler ihtiyaç halinde yatay olarak ölçeklenebilmelidir.

Mikro Hizmet Mimarilerde Dikkat Edilmesi Gerekenler

- **Veri Yönetimi ve Tutarlılık**
 - Her mikro hizmet kendi veritabanına sahip olmalı (Database per service).
 - Veri tutarlığı için eventual consistency, outbox pattern veya saga pattern gibi yöntemler tercih edilmeli.
- **API Tasarımı**
 - Mikro hizmetler arasındaki iletişim için net, tutarlı ve iyi dokümantel edilmiş API'ler oluşturulmalı.
 - REST, gRPC, GraphQL gibi uygun iletişim protokollerleri seçilmeli.
 - API versiyonlama ve geriye dönük uyumluluk stratejileri belirlenmeli
- **İletişim ve Mesajlaşma**
 - Senkron (HTTP/gRPC) ve asenkron (message broker - Kafka, RabbitMQ) iletişim dengeli kullanılmalı.
 - Mesajların güvenilir iletimi için retry, dead-letter queue mekanizmaları olmalı.

Mikro Hizmet Mimarilerde Dikkat Edilmesi Gerekenler

• Servis Keşfi ve Yönlendirme (Service Discovery & Routing)

- Mikro hizmetlerin konumları dinamik olduğundan servis keşfi mekanizmaları (Consul, Eureka, etc.) kullanılmalı.
- API Gateway veya Service Mesh (Istio, Linkerd) ile trafigin yönetilmesi sağlanmalıdır.

• Gözlemlenebilirlik (Observability)

- Merkezi logging, monitoring (Prometheus, Grafana) ve tracing (Jaeger, Zipkin) sistemleri kurulmalıdır.
- Loglar, metrikler ve izler sayesinde sistemin durumu ve hatalar kolayca takip edilmeli.

• Hata Yönetimi ve Dayanıklılık

- Circuit breaker, fallback, retry gibi desenler uygulanmalıdır.
- Sağlık kontrolleri (health checks) ve otomatik iyileştirme mekanizmaları olmalıdır.

Mikro Hizmet Mimarilerde Dikkat Edilmesi Gerekenler

• Güvenlik

- Hizmetler arası iletişim güvenli (TLS) olmalı.
- Kimlik doğrulama ve yetkilendirme merkezi (OAuth2, JWT, mTLS) ile yönetilmeli.
- Veri gizliliği ve servisler arası erişim kontrolleri kesinlikle uygulanmalı.

• Konfigürasyon Yönetimi

- Merkezi konfigürasyon yönetimi (Consul, Spring Cloud Config, Vault) kullanılmalı.
- Environment Variables (dev, test, prod) özel konfigürasyonlar dinamik ve güvenli olmalı.

• Teknoloji ve Dil Çeşitliliği

- Mikro hizmetler farklı teknoloji veya programlama dilleri ile geliştirilebilir.
- Ancak karmaşıklığı artırmamak için teknolojik uyumluluk ve standardizasyon dengelenmeli.

• Performans ve Kaynak Yönetimi

- Servislerin kaynak kullanımı ve performansı sürekli izlenmeli.
- Gecikme (latency) ve throughput değerleri optimize edilmeli.

Mikro Hizmet Mimarilerde Temel Kavramlar

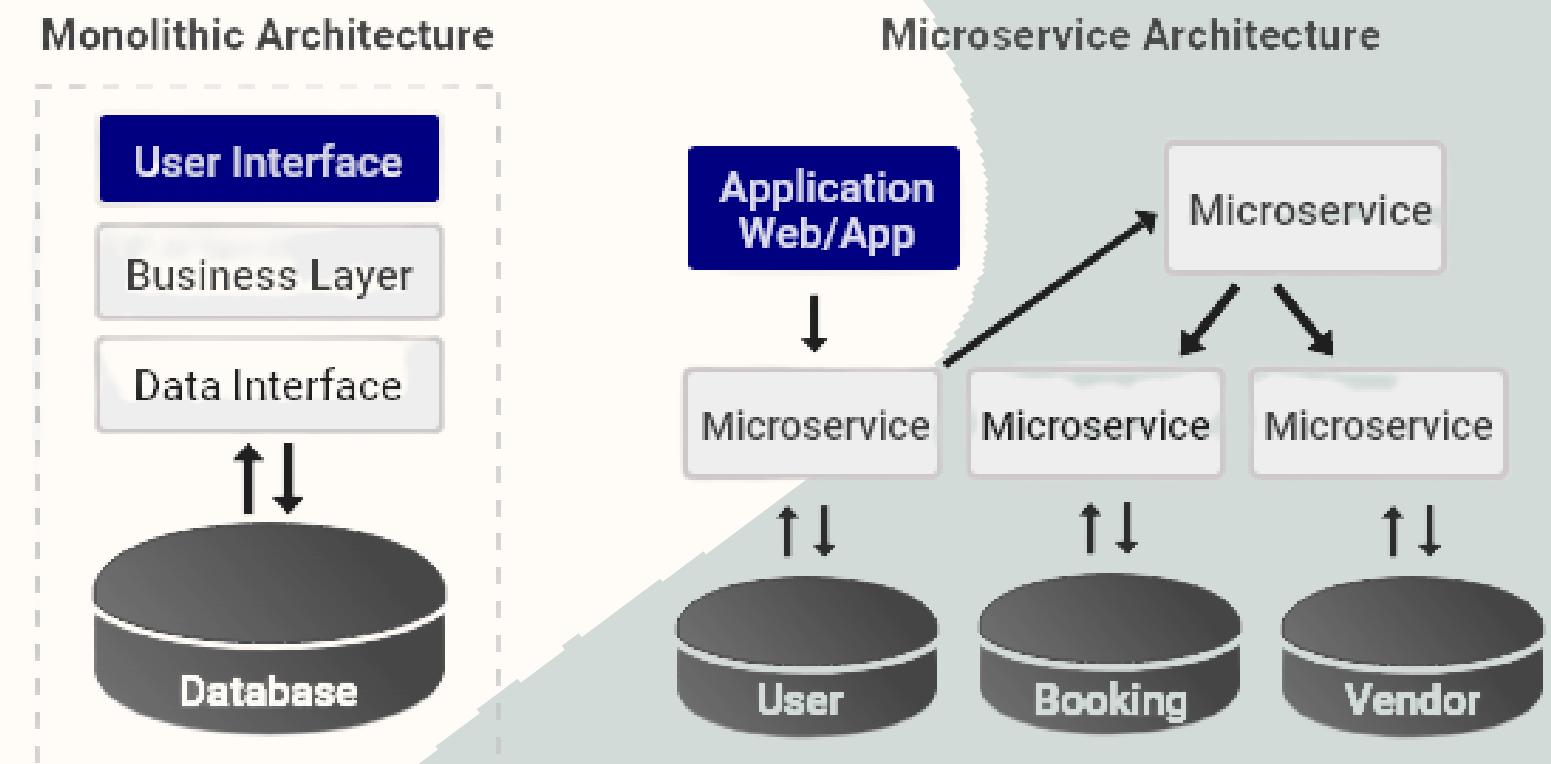
- **API Gateway**
 - Mikro hizmetlere dışarıdan gelen tüm isteklerin karşılandığı tek giriş noktası.
- **Service Discovery (Servis Keşfi):**
 - Mikro hizmetlerin birbirini dinamik olarak bulmasını sağlayan mekanizma.
- **Event-Driven Architecture (Olay Tabanlı Mimari):**
 - Servislerin birbirleriyle asenkron olarak olaylar (event) üzerinden haberleştiği yapı.
- **Message Broker**
 - Mikro hizmetlerin asenkron mesajlaşmasını sağlayan sistem (örn. Kafka, RabbitMQ).
- **Containerization (Konteynerleştirme):**
 - Mikro hizmetlerin bağımsız olarak paketlenip çalıştırılmasını sağlayan teknoloji (örn. Docker).
- **Service Mesh:**
 - Mikro hizmetler arasındaki iletişimini yönetmek için kullanılan altyapı katmanı (örn. Istio).

Mikro Hizmet Mimarileri

Mikro hizmet mimarisi, yazılım sisteminin birbirinden bağımsız, küçük, dağıtık servisler halinde geliştirildiği, konuşlandırıldığı ve çalıştırıldığı bir mimari yaklaşımıdır.

Her mikro servis:

- Tek bir iş sorumluluğu taşır.
- Kendi veritabanına sahip olabilir.
- Kendi başına deploy edilebilir.
- Diğer servislerle genellikle **HTTP**, **gRPC**, **AMQP** gibi protokoller üzerinden haberleşir



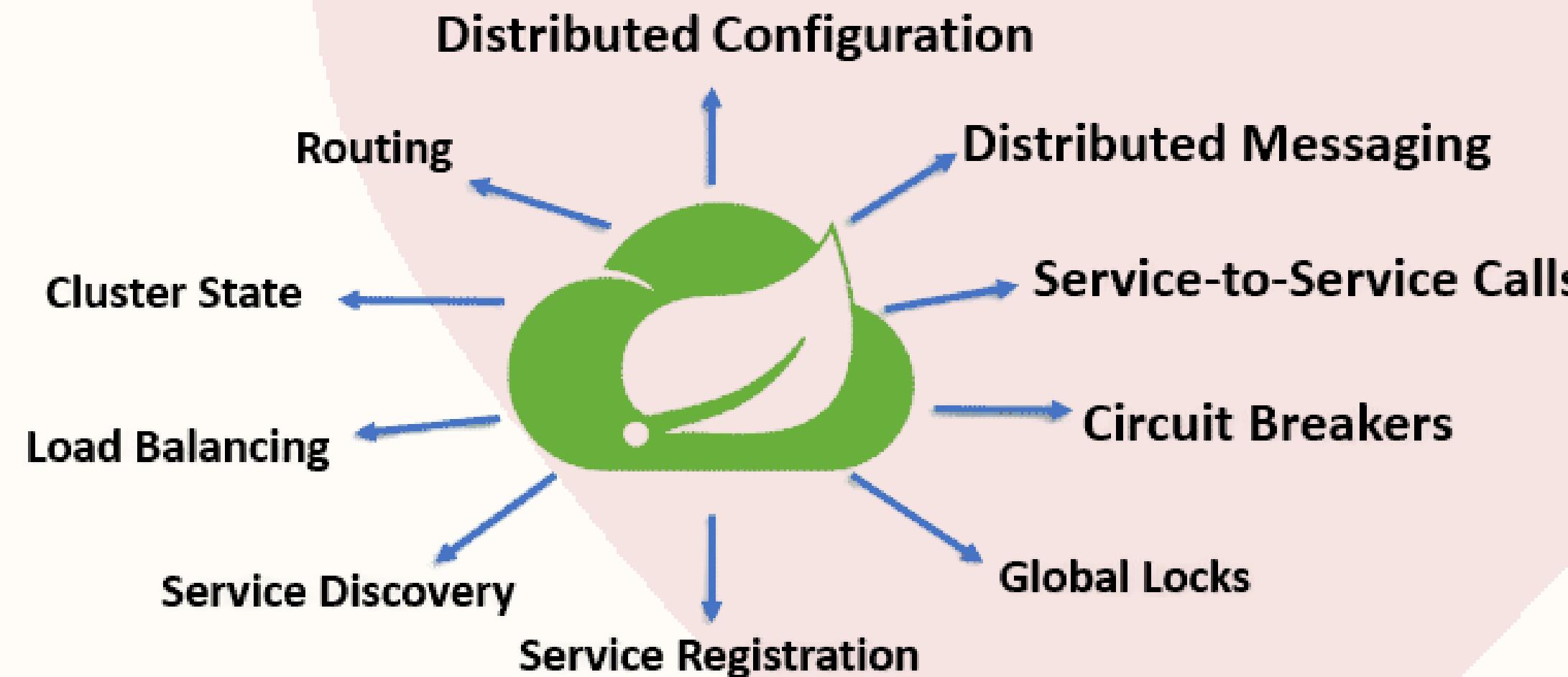


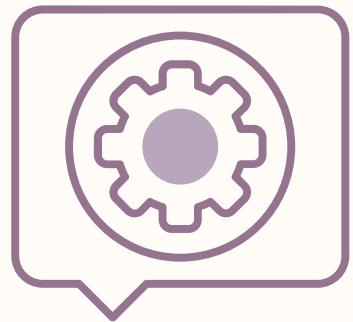
Spring Cloud

Spring Cloud, Spring Boot üzerine kurulmuş, dağıtık sistemler ve mikroservis mimarileri için çeşitli altyapı çözümleri sunan bir framework'tür.

Spring Cloud Nedir ?

Spring Cloud, dağıtık sistemler ve mikroservis mimarileri için bir araç ve kütüphane setidir. Spring ekosistemini kullanarak mikroservisler arası iletişim, konfigürasyonu, güvenliği ve yönetimi kolaylaştırır.





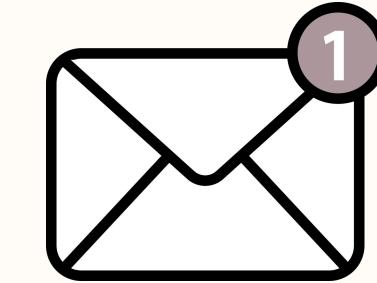
Spring Cloud Config

Merkezi konfigürasyon yönetimi sağlar



Spring Cloud Gateway

API Gateway görevi görür. Routing, filtreleme, rate limiting gibi işlemleri yapar.



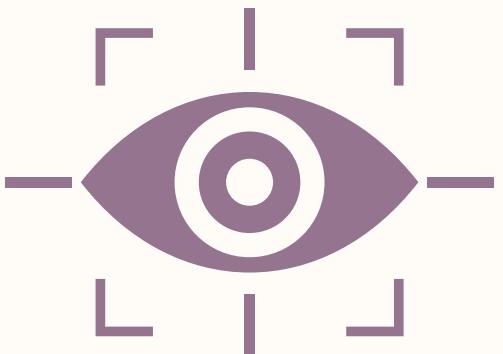
Spring Cloud Stream

Kafka, RabbitMQ gibi mesajlaşma altyapıları ile event-driven sistemleri destekler.



Eureka / Service Discovery

Mikroservislerin birbirini bulmasını sağlar. Servis kayıt ve keşif mekanizması sunar.



Spring Cloud Sleuth

Mikroservis çağrılarını izler ve trace bilgisi toplar.



Spring Cloud Load Balancer

Servisler arası isteklerde yük dengeleme sağlar.

Spring Cloud Config Server

Dağıtık Konfigürasyon Yönetimi

1 Config Server

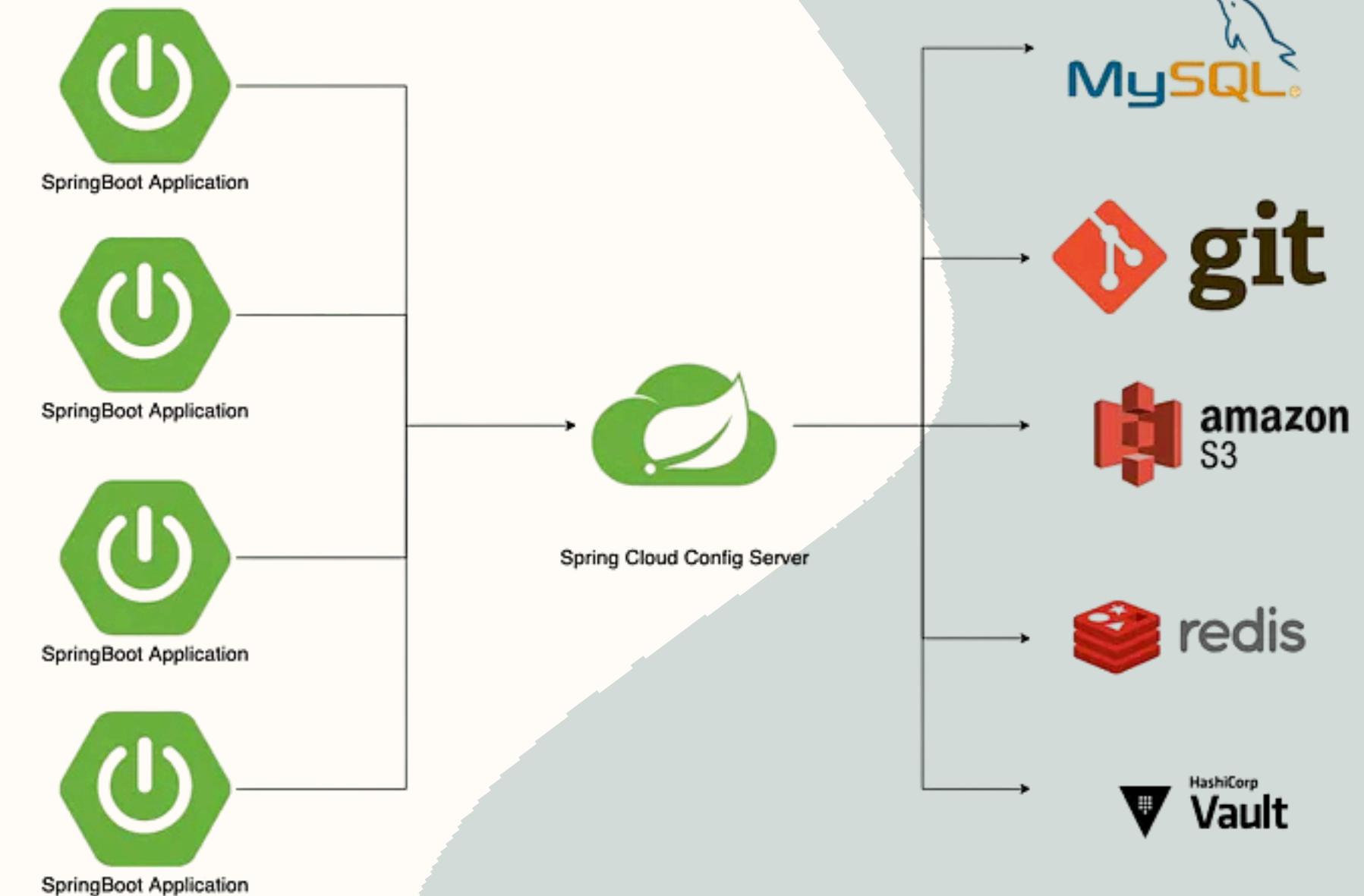
2 Config Client

3 Profiles & Labels

4 Cloud Config Server
Configurations

Spring Cloud Config Server

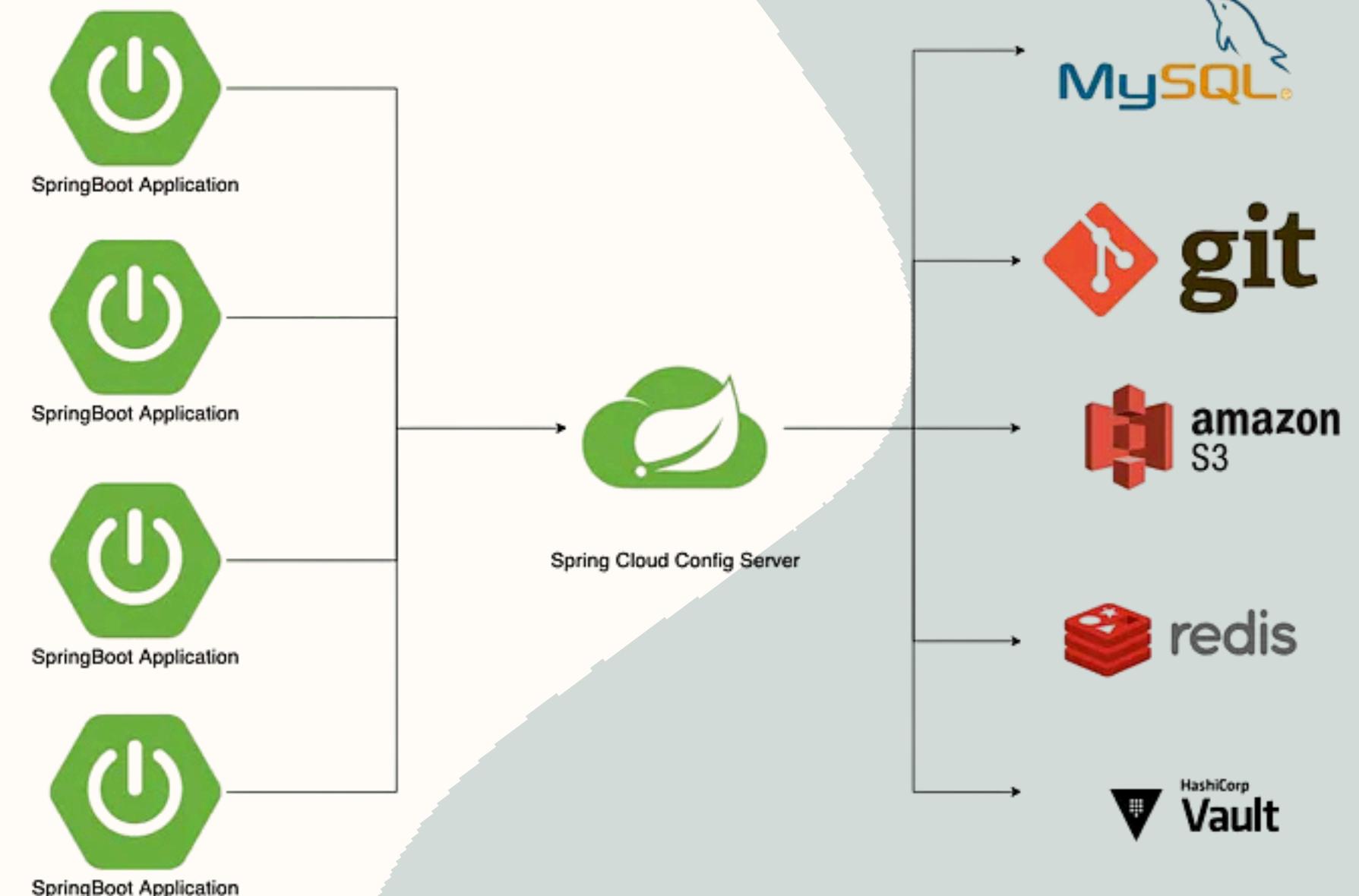
Cloud config server modülü; konfigurasyon dosyalarımızı bir dosya dizininden, git reposundan ya da farklı servisleri üzerinden(aws, hashicorp, cyberark vs.) okumamıza olanak sağlar.



Spring Cloud Config Client

Spring Cloud Config Client, Spring uygulamaları için merkezi konfigürasyon yönetimini sağlayan bir bileşendir. Basitçe, Config Server'dan konfigürasyonları çeken ve uygulamaya uygulayan istemci tarafıdır.

Config Serverdan çekilen konfigürasyonlar [@Value](#) veya [@ConfigurationProperties](#) anatasyonları kullanılarak okunabilir.



Spring Cloud Config Server Classpath

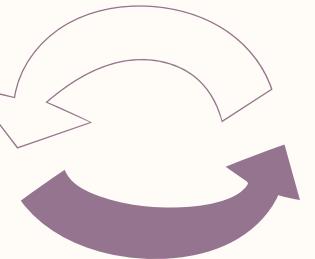
Konfigürasyon dosyalarının uygulama içerisinde classpath tabanlı yönetilmesini sağlar. Küçük ölçekli mikro hizmet mimarileri için tercih edilebilir.

Spring Cloud Config Server Git Repo

Konfigürasyon dosyalarının Git Repo üzerinden yönetilmesini sağlar.
çok fazla mikro hizmetin bulunduğu kurumsal projelerde tercih edilir.

Konfigürasyon dosyaları git, cyberark ya da vault servisleri ile
ayarlanır.

Bağımlılıklar



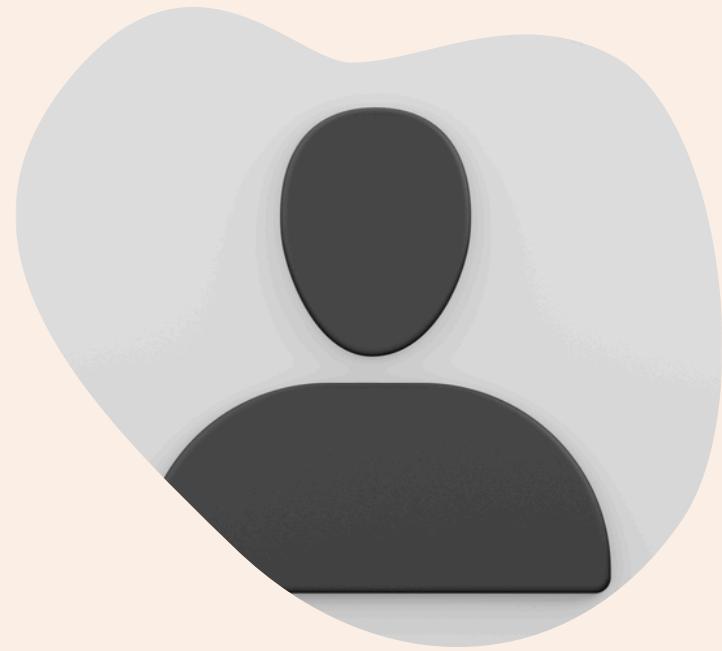
Config Server

org.springframework.cloud:spring-cloud-config-server

Config Client

org.springframework.cloud:spring-cloud-starter-config

Profiles & Label



Profiles

Aynı uygulama için farklı ortamlar (development, test, production) veya farklı koşullar altında farklı konfigürasyonlar sağlamak.



Labels

Config Server'ın konfigürasyon kaynağı olarak kullanılan Git (veya başka repository) üzerinde farklı versiyonları yönetmek.
master,develop,release/v1 branches

Konfigürasyon Yönetimi

Classpath

```
spring.application.name=config-server
server.port=8085
spring.profiles.active=native
spring.cloud.config.server.native.searchLocationsclasspath:/user-service-config/,classpath:/order-service-config/
```

Git Repo Config

```
# Cloud Server Config With Git Repo
server.port=8085
spring.application.name=conf-server
spring.profiles.active=git
spring.cloud.config.server.git.default-label=master
spring.cloud.config.server.git.uri=https://github.com/neominalolutions/spring-cloud-config-repo
spring.cloud.config.server.git.searchPaths=user-service-config/,order-service-config/
```

Spring Boot Admin

Spring Boot tabanlı mikro hizmetlerin yönetimi, izlenmesi ve görselleştirilmesi için kullanılan bir açık kaynak uygulama (Spring Boot Admin olarak da bilinir).

1 Mikro hizmetlerin durumu ve sağlık kontrolü

2 Servislerin kayıt ve keşfi

3 Performans ve metriklerin izlenmesi

4 Log takibi

5 Uyarı ve bildirimler

Spring Boot Admin

Spring Boot Actuator bileşeniyle entegre çalışır.

Spring Boot Admin, bir merkezi kontrol paneli (dashboard) sağlar.

Temel Bileşenler:

Spring Boot Admin Server: Tüm mikro servislerin kayıt olacağı merkezi yönetim paneli.

Spring Boot Admin Client: Mikro hizmetlerin, Admin Server'a kayıt olmasını sağlayan istemci konfigürasyonu.

Spring Boot Actuator: Spring Boot uygulamalarına sağlık, metrik, trace, log, env gibi uç noktaları (endpoint) ekler.

Not: Prod Ortamda Sınırlama `management.endpoints.web.exposure.include=*` yerine sadece gereken endpointleri açman önerilir.



Spring Boot Admin



Spring Boot Admin arayüzünde aşağıdakileri görüntüleyebilirsin:

- **Health**: Client uygulamaların sağlık durumu (up, down)
- **Metrics**: JVM, memory, GC, request count gibi performans verileri
- **Loggers**: Log seviyelerinin canlı değişimi
- **Environment**: Client uygulamaların env değişken tanımları
- **Beans**: Client uygulamasındaki Spring bean tanımları
- **HTTP Traces**: Son HTTP isteklerinin listesi
- **Thread Dump**: JVM thread bilgileri
- **JMX**: JMX MBean'ler
- **Mappings**: URL – Controller mapping bilgileri

Not: Spring Boot Admin, daha çok uygulama sağlığı ve operasyonel görünürlük için uygundur. Eğer log analiz, alerting, distributed tracing gibi ihtiyaçlar varsa Zipkin, ELK Stack gibi araçlardan yararlanılabilir.

Spring Boot Admin



Spring Boot Admin, olaylar (event) için bildirim sistemi entegre edebilir:

Desteklenen Bildirim Kanalları:

Email (SMTP), Slack, PagerDuty, Microsoft Teams, HipChat, OpsGenie, Webhook

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

Bağımlilikler

Spring Boot Admin Server

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
```

Spring Boot Admin Client

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Konfigürasyon

```
spring:  
  application:  
    name: my-microservice  
  boot:  
    admin:  
      client:  
        url: http://localhost:8080 # Spring Admin Url  
    instance:  
      prefer-ip: true  
  management:  
    endpoints:  
      web:  
        exposure:  
          include: "*" # ile actuator endpoint'lerinin hepsi açılır. Prod  
ortamda dikkatli olunmalıdır
```

Eureka Entegrasyonu

Spring Cloud ortamlarında servis keşfi için Eureka kullanıyorsan, Admin Server, Eureka'ya kayıtlı servisleri otomatik algılayabilir.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

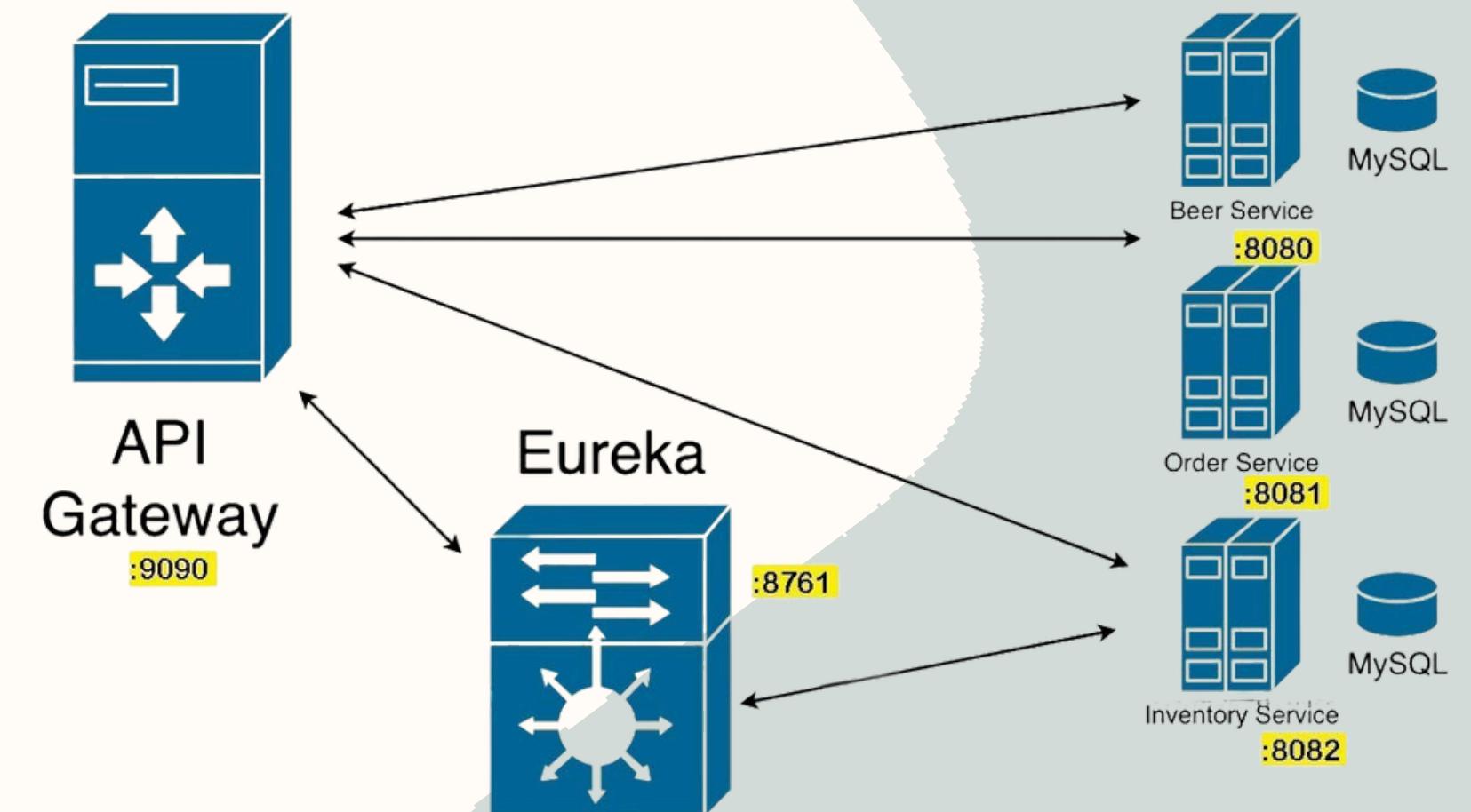
```
spring:
  boot:
    admin:
      discovery:
        enabled: true
```

Service Discovery

Microservislerin birbirlerini otomatik olarak bulmasını ve çağrımasını sağlayan merkezi bir kayıt mekanizmasıdır.

Amaç:

- Servislerin statik IP veya port bilgisi olmadan birbirini çağrıramaması
- Ölçeklenebilirlik ve esneklik sağlamak (örn. servis sayısı arttığında manual değişiklik yok)



Service Discovery

Kullanım Alanı:

- Genellikle internal (iç) servisler arasında kullanılır, yani microservice'lerin kendi aralarında iletişim kurmasını sağlar.
- API Gateway veya external client erişimi için de kullanılabilir ama genellikle internal discovery yapılır.

Nasıl Çalışır?

1. Her servis kendini service registry'ye kaydeder (register).
2. Başka bir servis, registry üzerinden servis adını kullanarak IP/port bilgisine ulaşır.
3. Load balancing ve fault tolerance için ek katmanlarla birlikte çalışabilir.



Netflix Eureka

Netflix'in open-source olarak sunduğu Service Registry ve Discovery çözümüdür.

Ana Bileşenler:

- Eureka Server:** Merkezi registry, servis kayıtlarını tutar.
- Eureka Client:** Servislerin server'a kaydolmasını ve diğer servisleri bulmasını sağlar.

Özellikler;

- Servisler dinamik olarak kaydolur ve silinir
- Heartbeat mekanizması ile servis sağlığı kontrolü
- Spring Boot + Spring Cloud ile kolay entegrasyon
- Cluster kurulumunda yüksek erişilebilirlik sağlar

Konfigürasyon Yönetimi

Eureka Server

```
#Eureka Server Settings
spring.application.name=eureka-server
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Eureka Client

```
#Eureka Client Settings
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```

Bağımlilikler

Eureka Server

org.springframework.cloud:spring-cloud-starter-netflix-eureka-server

Eureka Client

org.springframework.cloud:spring-cloud-starter-netflix-eureka-client

Mikro Hizmet Mimarilerde İletişim

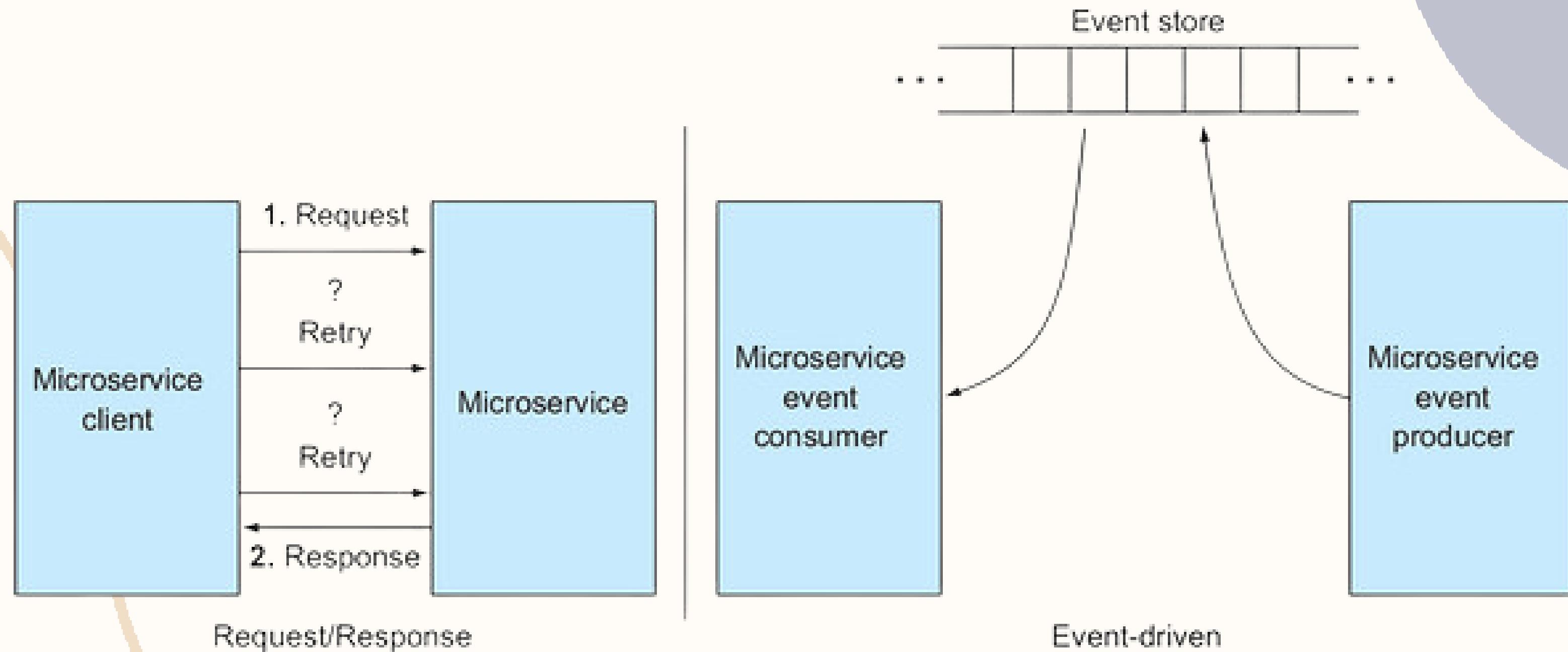
Request Driven: Servisler birbirine doğrudan çağrı yapar (HTTP/REST, gRPC)

- Senkron çalışır ama gecikmeye ve tight coupling'e yol açabilir
- Kolay anlaşılır, debug kolay
- Ama gecikmeye ve tight coupling'e yol açabilir

Event Driven: Servisler event publish/subscribe ile haberleşir (Kafka, RabbitMQ).

- Asenkron çalışır ve Servisler loosely coupled, daha esnek ve ölçeklenebilir
- İzleme ve hata yönetimi daha karmaşık olabilir

Request & Event Driven



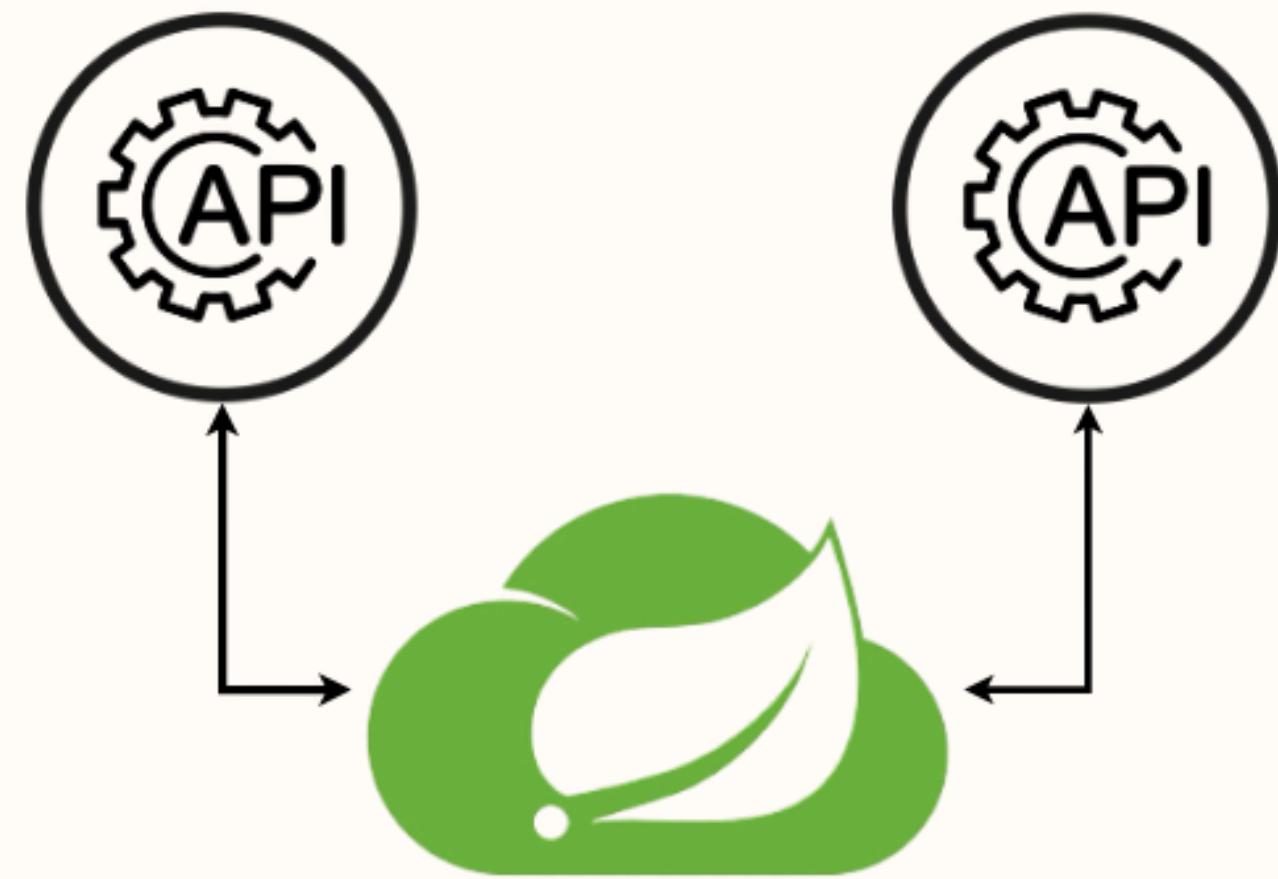
Feign Client

Feign Client, bir servisi çağrılmak için interface kullanarak kodu basitleştiren ve declarative HTTP client sağlayan bir Spring Cloud bileşenidir.

Servislere interface üzerinden HTTP çağrıyı yapmayı sağlar.

Kodda endpoint URL'leri hard-code etmeden, servis isimleri ile çağrı yapılabilir.

Mikro hizmetler arası Request Driven
Mimaride Feign Client kullanınız

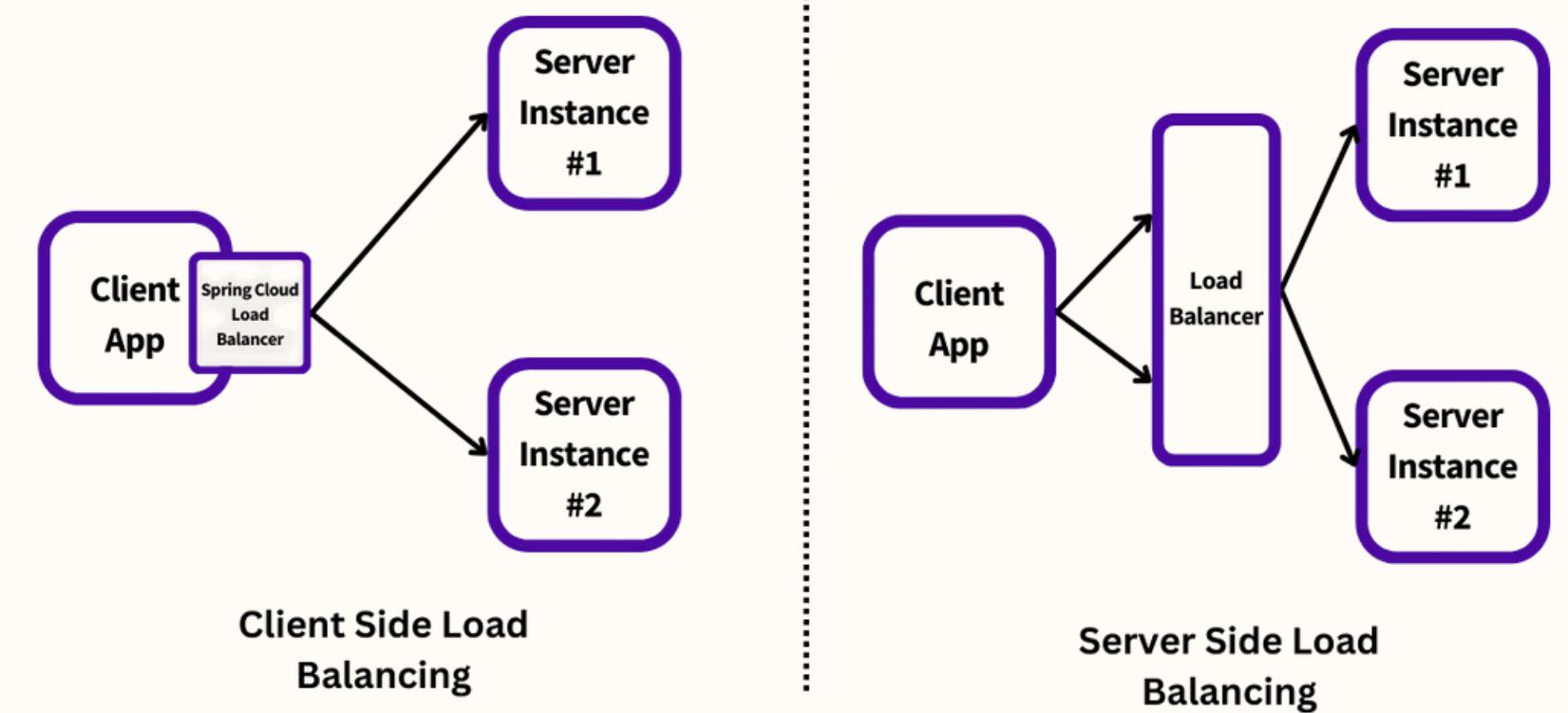


Feign Client

Spring Cloud Load Balancer

Mikroservis mimarisinde servisler arası iletişimde yük dengeleme (load balancing) yöntemiyle daha verimli hale getiren, Spring Cloud ekosistemine entegre bir bileşendir.

Not: Eskiden Netflix Ribbon kullanılıyordu. Ancak Ribbon artık deprecated (kullanımdan kalktı). Yerine Spring Cloud LoadBalancer geldi.



Spring Cloud Load Balancer

Spring Cloud LoadBalancer, genellikle Spring Cloud Gateway, OpenFeign veya RestTemplate ile entegre şekilde çalışır.

Spring Cloud LoadBalancer, varsayılan olarak **Round Robin** stratejisini kullanır.

- Round Robin → Sırayla her instance'a istek gönderir.
- Random → Rastgele bir instance seçer.
- Custom → Kendi yazdığımız strateji.

Not: Eureka ile birlikte LoadBalancer kullanmak, mikroservislerin IP adreslerine ya da portlarına sabit olarak bağlı kalmadan çalışmasını sağlar. Dinamik ve esnek yapı elde edilir.

Feign + Eureka + LoadBalancer birlikte çalışarak servisler arası çağrıları dinamik, yük dengeli ve basit hale getirir.

Bağımlilikler

```
<!-- Eureka client -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<!-- Load Balancer -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
<!-- Feign Client (isteğe bağlı) -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Mikro Hizmetlerde Resilincy

Bir mikroservis uygulamasının hatalara karşı dayanıklı olması, hatalar meydana geldiğinde çökmeden ayakta kalabilmesi anlamına gelir.

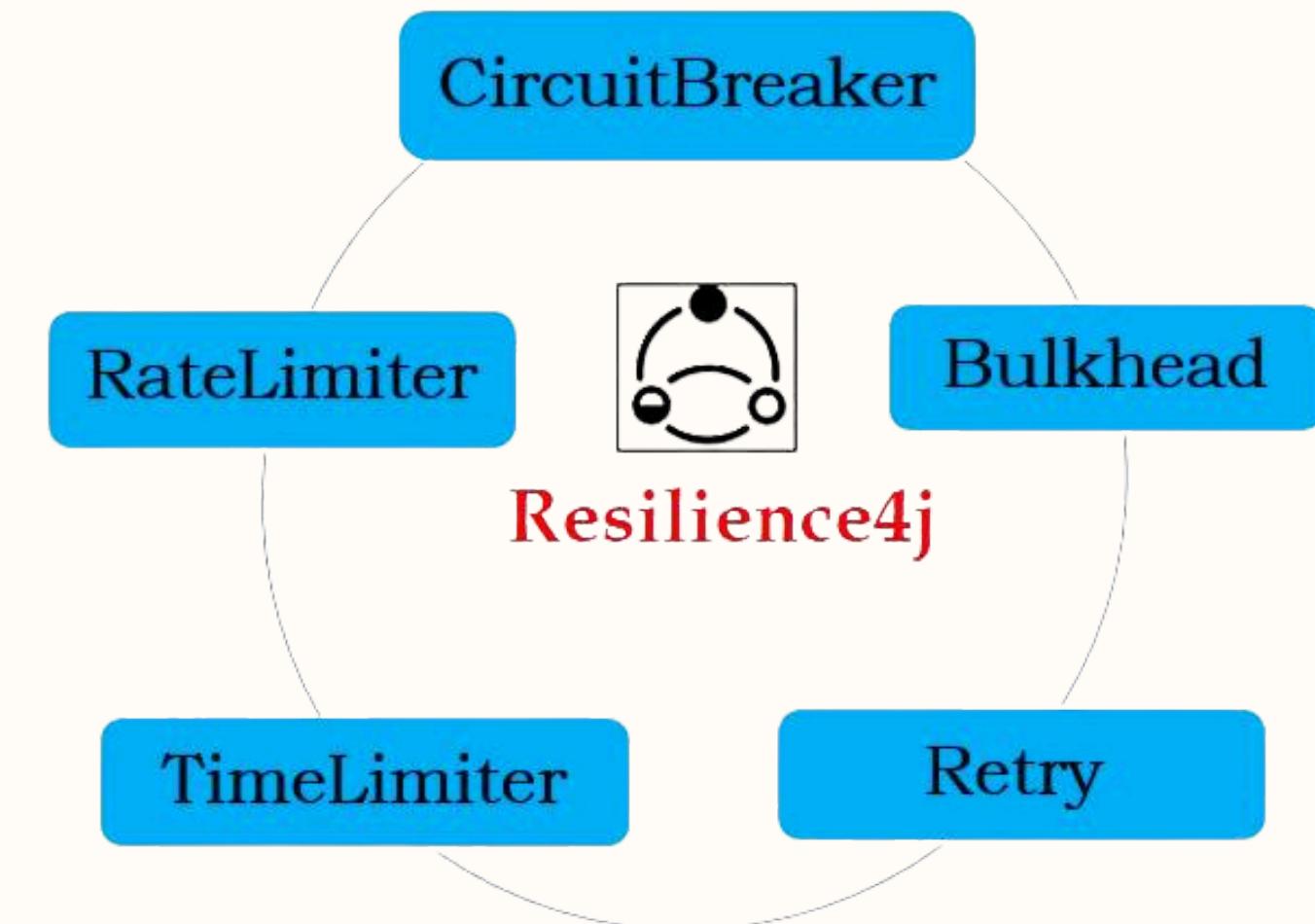
Neden Önemlidir?

Mikroservisler dağıtık sistemlerdir. Bu sistemlerde:

- Ağ hataları,
- Zaman aşımıları,
- Servis çökmesi,
- Yoğun trafik gibi durumlar kaçınılmazdır.

Amaç,

- Sistemin tamamen çökmesini engellemek,
- Kullanıcı deneyimini korumak,
- Hataların sistemde zincirleme etki yaratmasını önlemektir.



Spring Boot Resilience4j

Mikroservislerde kullanılmak üzere geliştirilmiş hafif, modüler, fonksiyonel ve Java 8+ uyumlu bir resiliency kütüphanesidir.

Not: Netflix'in eski Hystrix kütüphanesinin yerini almıştır ve Spring Boot ile mükemmel uyum sağlar.

Önemli: RateLimiter → Bulkhead → TimeLimiter → CircuitBreaker → Retry → Service Method

Circuit Breaker: Servis hatalarında trafigi keserek sistemin toparlanmasına izin verir.

Rate Limiter: Servise belirli bir süre içinde yapılan istek sayısını sınırlar.

Retry: Serviste hata oluşunca isteği tekrar deneme mekanizması.

Bulkhead: Paralel çağrı sayısını sınırlı olarak kaynak tüketimini kontrol altına alır.

TimeLimiter: Belirli bir sürede cevap veremeyen servisleri keser.

Circuit Breaker Pattern

- Sürekli hata veren bir servise istek göndermeyi durdurur.
- Sistem yükünü azaltır.
- Sorun geçince tekrar denemeye başlar.

CLOSED: Her şey normal, istekler normal şekilde gönderiliyor.

OPEN: Hatalar arttı, artık istekler engelleniyor.

HALF-OPEN: Test için az sayıda istek gönderiliyor, eğer başarılıysa tekrar CLOSED olur.

Circuit Braker Config

```
resilience4j:  
    circuitbreaker:  
        instances:  
            orderService:  
                registerHealthIndicator: true # Actuator üzerinden circuit breaker durumu izlenebilir  
                slidingWindowSize: 5          # Son 5 çağrı değerlendirilir  
                minimumNumberOfCalls: 3      # Circuit breaker karar vermeden önce en az 3 çağrı olmalı  
                failureRateThreshold: 50     # %50 başarısızlık oranına ulaşılırsa devre kesilir (open state)  
                waitDurationInOpenState: 5s   # Open state'te kaldıktan sonra 5 saniye bekleyip tekrar deneme yapılır
```

Retry Pattern

Geçici hatalarda aynı işlemi tekrar deneyerek başarı şansını artırır.

Not: Varsayılan olarak 3 kez dener ve her deneme arasında bekler.

```
retry:  
  instances:  
    orderService:  
      max-attempts: 3 # Maksimum deneme sayısı (ilk istek + 2 retry = toplam 3)  
      wait-duration: 1s # Her deneme arasında 1 saniye bekle
```

Rate Limitter Pattern

Belirli bir sürede yapılan istek sayısını sınırlar (örneğin saniyede 10 istek).

Not: Anı yüklenmeleri engelleyerek sistemin çökmesini önler.

```
ratelimiter:  
  instances:  
    orderService:  
      limit-for-period: 5  # 100s 5 adet çağrı limiti  
      limit-refresh-period: 100s  
      timeout-duration: 0s
```

Bulkhead Pattern

Mikroservis sistemlerinde eş zamanlı çalışan işlemleri sınırlamak için kullanılır. Bu sayede bir servis aşırı yük altında kaldığında diğer servislerin etkilenmesini engeller.

Thread-based (ThreadPoolBulkhead): Farklı bir thread havuzu kullanarak izolasyon sağlar.

```
@Bulkhead(name = "myService", type = Bulkhead.Type.ThreadPoolBulkhead)
```

Semaphore-based: Belirli sayıda eşzamanlı işlemi semafor ile sınırlar. Daha hafif, CPU dostu. (Varsayılan ayar)

```
bulkhead:
  instances:
    orderService:
      max-concurrent-calls: 3 # Aynı anda en fazla 3 thread bu servise çağrı yapabilir
      max-wait-duration: 0s   # Eğer thread havuzu dolussa, beklemeden hata ver (0 ms bekle)
```

TimeLimiter Pattern

Bir metodun belirli bir süre içinde cevap vermesini bekler. Süre aşılırsa, işlemi keser ve fallback'a düşer. Bu sayede "tutulup kalan" threadler sistemi kilitlemez.

Not: TimeLimiter sadece asenkron işlemlerde -**CompletableFuture**- çalışır.

```
timelimiter:  
  instances:  
    orderService:  
      timeout-duration: 2s          # Metot 2 saniye içinde tamamlanmazsa iptal edilir.  
      cancelRunningFuture: true # süre aşımı olursa arka plandaki işlemleri de durdur
```

Bağımlilikler

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot3</artifactId>
</dependency>
```

Not: Resilience4j, Actuator ile entegre çalışarak durumu gözlemebilmeni sağlar.

TimeLimiter Pattern

Bir metodun belirli bir süre içinde cevap vermesini bekler. Süre aşılırsa, işlemi keser ve fallback'a düşer. Bu sayede "tutulup kalan" threadler sistemi kilitlemez.

Not: TimeLimiter sadece asenkron işlemlerde -**CompletableFuture**- çalışır.

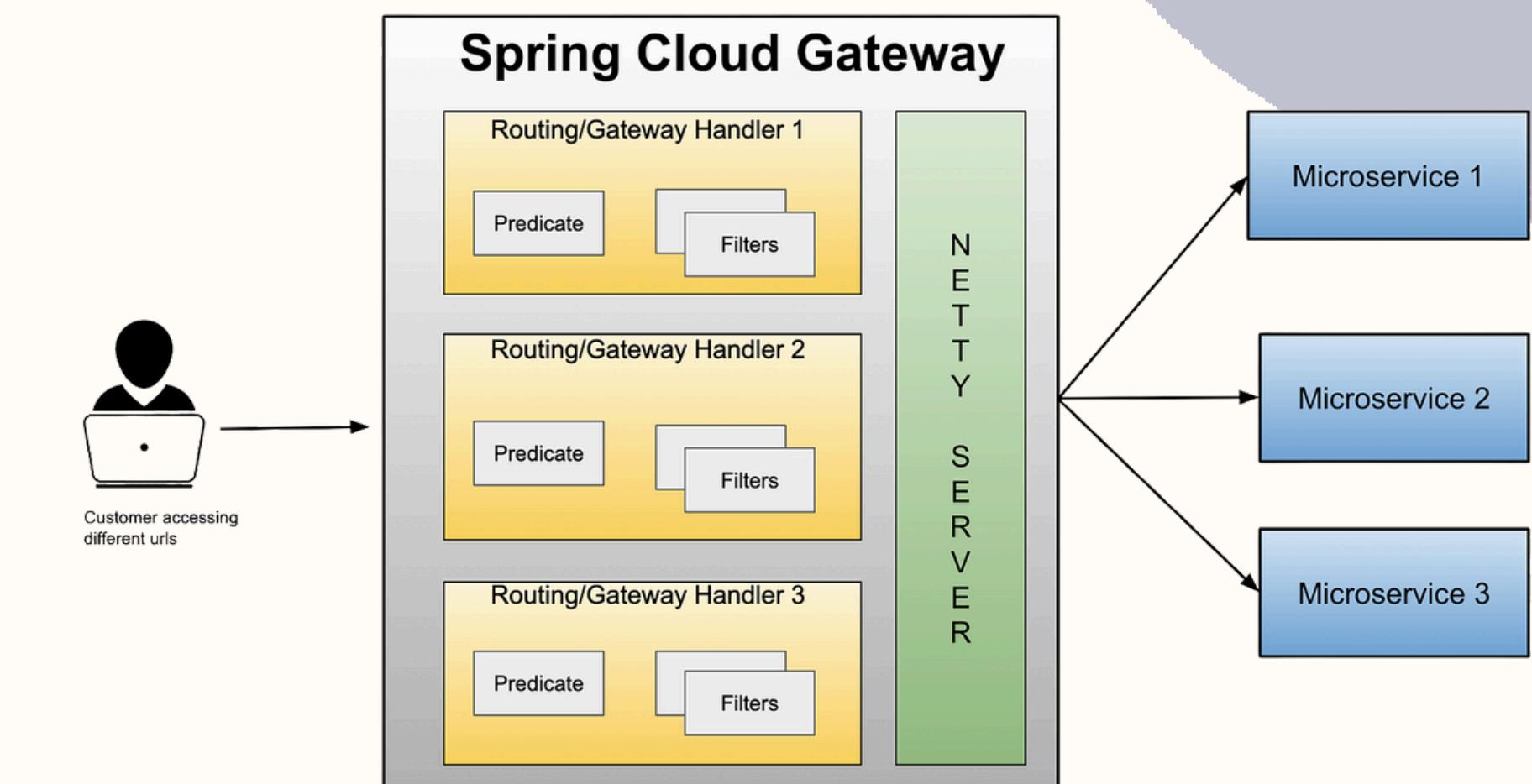
```
timelimiter:  
  instances:  
    orderService:  
      timeout-duration: 2s          # Metot 2 saniye içinde tamamlanmazsa iptal edilir.  
      cancelRunningFuture: true # süre aşımı olursa arka plandaki işlemleri de durdur
```

API Gateway

API Gateway, mikro hizmetlerin önünde duran, tüm dış talepleri alan ve uygun mikro hizmetlere yönlendiren tek giriş noktasıdır.

İstemciler (web, mobil vs.) doğrudan mikro hizmetlerle konuşmak yerine, tüm isteklerini API Gateway'e gönderir.

API Gateway, istekleri uygun mikro hizmete yönlendirir, istek ve yanıtları işler.



Spring Web Flux

Spring WebFlux, Spring Framework'ün reactive (tepkisel) web uygulamaları geliştirmek için sunduğu modülüdür. WebFlux, HTTP (ve WebSocket) protokollerini üzerinden çalışan bir reactive web framework'tür.

Özellikleri:

- Non-blocking, asynchronous (eş zamansız) bir programlama modeline dayanır.
- Servlet API yerine Reactive Streams API kullanır, ve Backpressure (yük yönetimi) desteğiyle sistem kaynakları daha iyi kontrol edilir.
- Reaktif sunucu desteği vardır. Tomcat yerine, Netty gibi reactive sunucularla çalışır.
- Mono ve Flux tipinde asenkron yapıları destekler.
- Functional veya annotation tabanlı geliştirilebilir.

Not: Spring Cloud Gateway, reactive bir yapıda çalışmalıdır. Yüz binlerce istekle tek bir merkezden başa çıkabilmesi için non-blocking yapıları desteklemelidir.

Spring Cloud Gateway

Spring Cloud Gateway, reactive, non-blocking mimariye sahip, yüksek performanslı API Gateway çözümüdür. Bu yüzden **Spring WebFlux** üzerine inşa edilmiştir.

Avantajları

- Merkezi güvenlik ve kimlik doğrulama (auth) uygulama
- Tek noktadan rate limiting, throttling, logging
- CORS, load balancing, fallback, circuit breaker entegrasyonu
- API versiyonlama ve istek yönlendirme kolaylığı
- Mikro hizmetlerin istemciden gizlenmesi, soyutlanması sağlanır

Not: Gateway üzerinde security konfigürasyonu yaparken Spring Security'nin reactive modülü **@EnableWebFluxSecurity** kullanılır.

Spring Cloud Gateway

Başlıca özellikler:

- **Routing:** HTTP isteklerini URI, header, parametre vb. koşullara göre farklı backend servislere yönlendirme
- **Predicate ve Filter:**
 - **Predicate:** İsteklerin yönlendirme için koşul kontrolü
 - **Filter:** İstek/yanıt üzerinde işlem yapma (header ekleme, güvenlik kontrolü, logging, rate limiting)
- **Load Balancing:** Yük dengeleme için Spring LoadBalancer desteği
- **CORS Desteği:** Cross-Origin isteklerini yönetme

Spring Cloud Gateway

Başlıca özellikler:

- **Güvenlik:** OAuth2 Resource Server yapılandırması, JWT doğrulama, Merkezi kimlik doğrulama mekanizması
- **Service Discovery:** Eureka, Consul gibi registry'lerle entegre çalışma
- **Fallback ve Retry:** Hata durumunda alternatif rota veya yeniden deneme mekanizması

Spring Cloud Gateway Routing

2 farklı yapılandırma destekler;

- **Configürasyon Bazlı Routing** (spring-cloud.version=2023.0.0 ve öncesi)
 - application.yml veya application.properties dosyasına route tanımları yazılır.
- **Fonksiyon Bazlı Routing** (spring-cloud.version=2025.0.0 ve sonrası)
 - Spring WebFlux fonksiyonel yapısını kullanarak route'lar programatik olarak tanımlanır.

Bağımlılıklar

Spring Cloud Gateway:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Spring WebFlux:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Spring Load Balancer Entegrasyonu

Spring Cloud Gateway, mikro hizmetlerin örneklerine istekleri load balancer aracılığıyla yönlendirir. Bağımlılık olarak Spring Cloud Gateway projelerine eklenmesi gereklidir.

Not: **Spring LoadBalancer** kütüphanesi, Ribbon'un yerini almış hafif, reaktif uyumlu bir load balancer'dır.

Not: Gateway ile birlikte kullanıldığında, **lb://user-service** URI'si, Eureka veya başka discovery'den aldığı servis örneklerine otomatik yönlendirme yapar.

Eureka ile Service Discovery Entegrasyonu

Gateway'de **IP** veya **port** bilgisi statik yazılmamalıdır, dinamik ve ölçeklenebilir yapıya uygun geliştirilmesi gereklidir.

Bu dinamikliği sağlamak için gateway serviceleri kendilerini Eureka Server'a register eder. Bu sayede servisler Spring Cloud Loadbalancer yardımı ile **lb://servis-adi** şeklinde keşfeder.

spring.cloud.gateway.discovery.locator.enabled: true

Bağımlılıklar

Spring Cloud Eureka Client:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-eureka-client</artifactId>
</dependency>
```

Spring Cloud Loadbalancer:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

Identity Service Entegrasyonu

Gateway üzerinden geçen isteklerde kimlik doğrulama ve yetkilendirme için OAuth2 Resource Server konfigürasyonu yapılabilir. Böylece arkauç servisleri için merkezi bir kimlik doğrulama mekanizması kurulmuş olur.

- Gateway, gelen JWT tokenları doğrulama ve yetkilendirme kararları alabilir.
- **Spring Security** ile entegre çalışır.
- Gateway'de güvenlik filtreleri (**Security WebFilterChain**) tanımlanır, yetkisiz istekler reddedilir.

```
spring.security.oauth2.resource-server.jwt.issuer-uri=http://<domain>/realms/<realm>
```

Not: Keycloak'un kendi public key'lerini (JWKS) verdiği endpoint'tir. Jwt doğrulamaları bu key üzerinden yapılır. Eğer issuer-uri doğruysa, Spring otomatik olarak .well-known/openid-configuration üzerinden jwks_uri'yi bulur. Aşağıdaki yapılandırmaya gerek kalmaz.

```
spring.security.oauth2.resource-server.jwt.jwk-set-  
uri=http://<domain>/realms/<realm>/protocol/openid-connect/certs
```

Bağımlilikler

Oauth2 Resource Server:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

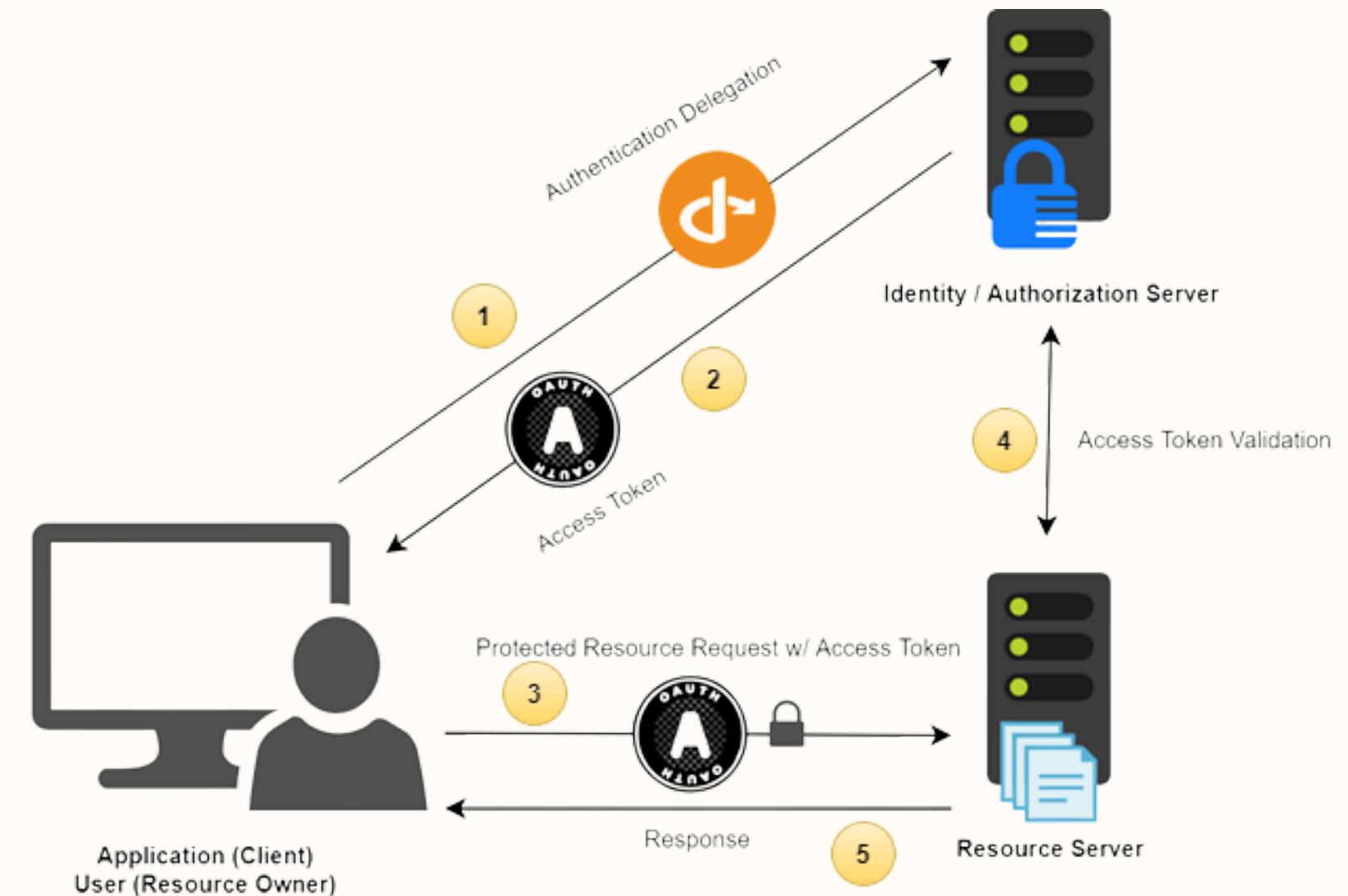
Spring Boot Security

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

OAuth2.0 & OpenId Connect

OAuth2.0, kullanıcının kimliğini ifşa etmeden (şifre vermeden), bir uygulamanın başka bir uygulamadaki kaynaklara erişmesini sağlayan bir yetkilendirme protokolüdür.

OIDC ise, OAuth 2.0 üzerine kurulu bir kimlik doğrulama (authentication) protokolüdür.



OAuth2.0 & OpenId Connect

Temel Akış:

- Kullanıcı, uygulamaya (Client App) giriş yapar.
- Uygulama, Authorization Server'a yönlendirir.
- Bu yönlendirme sırasında Client App şu bilgiler ile Authorization Server'a istek atar.
 - **client_id, redirect_uri, response_type=code, scope**
- Kullanıcı giriş yapar ve Authorization Serverdan **Authorization Code** alınır.
- Uygulama bu kodla **Access Token** alır.
- Uygulama, bu token ile resource server'a istek yapar.

OAuth2.0 & OpenId Connect

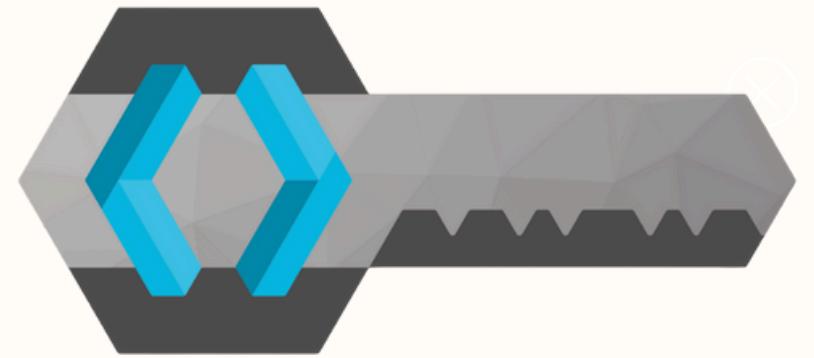
Resource Owner: Resource Server'a talepde bulunan kaynak sahibi, kullanıcı

Resource Server: Token ile erişilen API ya da servis

Client: Kaynağa erişmek isteyen uygulama

Authorization Server: Kimlik doğrulayıp Client ile Resource Server haberleşmesini yöneten sunucu.

- Authorization Code, Access Token, Refresh Token, ID Token gibi tokenları üretir.
- scope, aud, exp, roles gibi bilgileri token'a yazar
- Client uygulamasına bu token'ları verir
- Resource Server, bu token'ları doğrularken Authorization Server'ın public key'ini kullanır (JWKS)

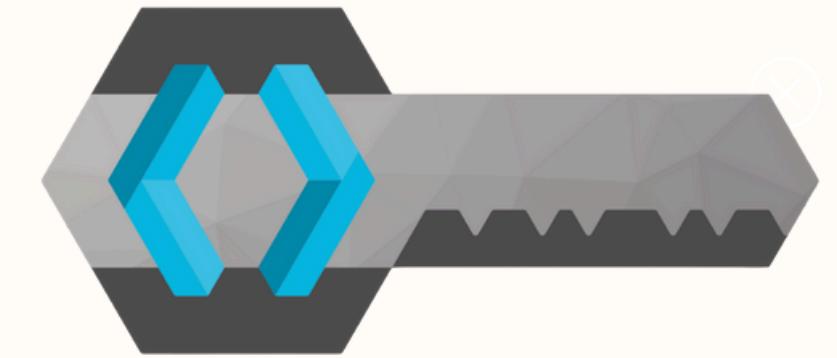


Keycloak Identity Server

Java ile yazılmış, açık kaynaklı bir **Identity and Access Management (IAM)** sistemidir.

- OAuth 2.0 ve OpenID Connect destekler.
- Spring Boot, Angular, React, mobil uygulamalar ile entegre olabilir.
- Arkasında Red Hat vardır.

Keycloak Identity Server



Realm: Ayrı bir güvenlik alanı. Her realm, kendi kullanıcılarını, rollerini ve istemcilerini yönetir.

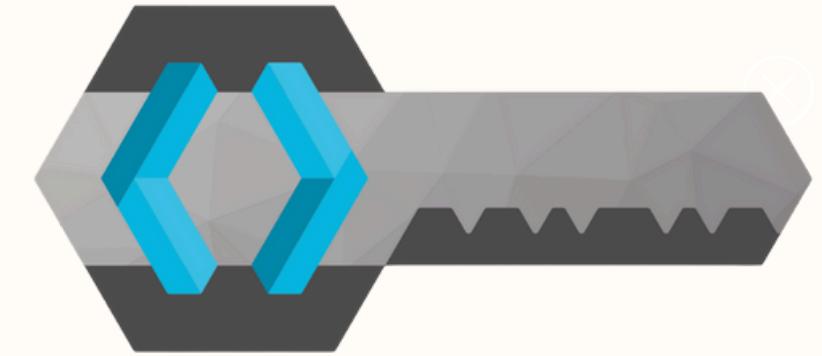
Client: Uygulamanın Keycloak ile tanımlı hali. (Spring app, frontend, vs.) Her client'in kendi ayarları vardır.

User: Sisteme giriş yapabilen kullanıcı. Keycloak UI üzerinden oluşturulur.

Role: Kullanıcılara veya client'lara atanabilen yetkiler.

Not: Kullanıcının bir servise erişimi için keycloak üzerinde **account client'a** role ataması yapmamız gereklidir. Keycloak da account client hesap erişimleri için açılmış ortak bir client görevi görür.

Keycloak Identity Server



Group: Roller bir araya getirilerek gruplar oluşturulabilir.

Identity Provider (IdP): Harici kimlik sağlayıcıları (Google, Facebook, LDAP vb.) ile giriş yapılmasını sağlar.

Protocol Mapper: Token içine özel claim'ler eklemek için kullanılır.

Token: Keycloak, access token, refresh token ve id token üretir.

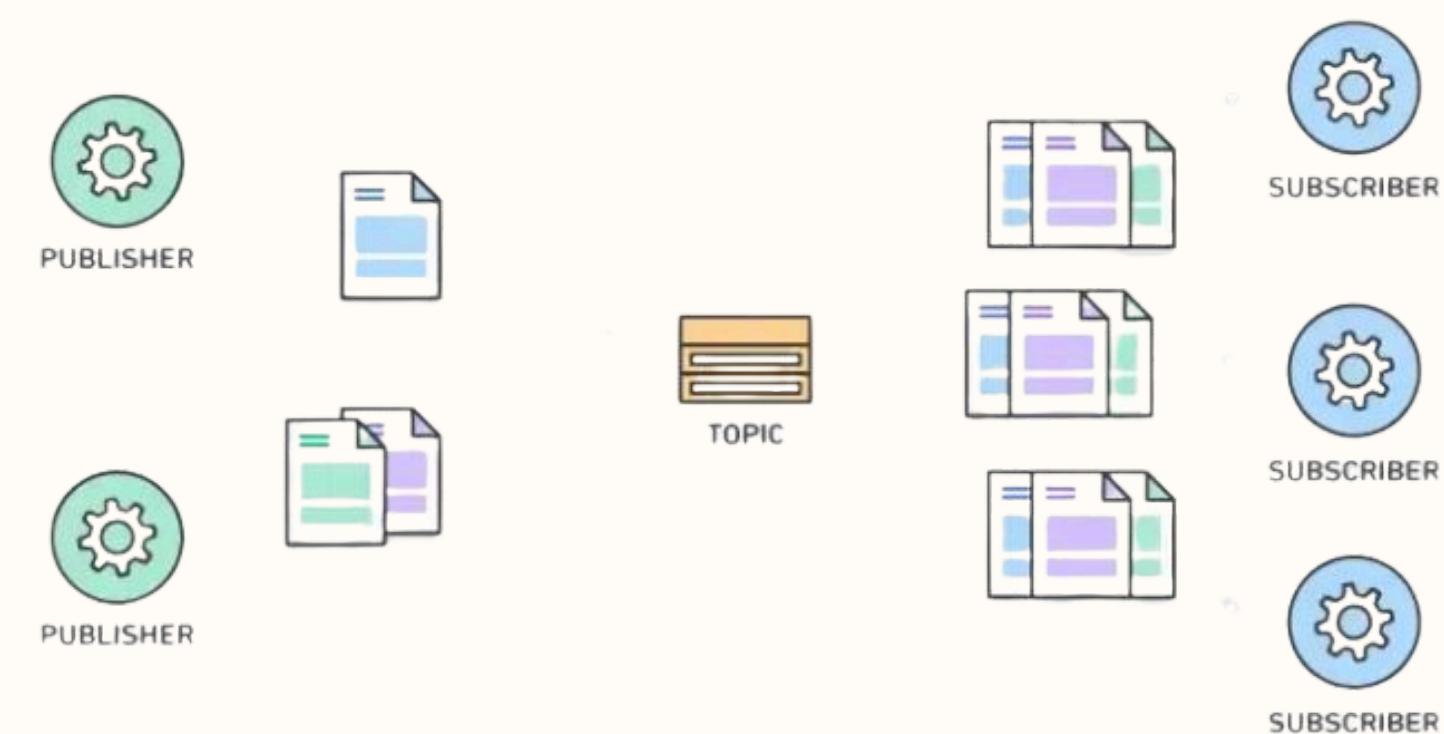
Pub/Sub Pattern

Amac: Servisler veya bileşenler arasında loosely coupled ve asenkron haberleşme sağlamak.

Publisher → Event veya mesaj yayınlar

Broker / Message Bus → Mesajı alır ve ilgili subscriber'lara iletir

Subscriber → Mesajı dinler ve işler



Pub/Sub Mimarisi

Spring Cloud Stream

Amac

- Kafka veya RabbitMQ ile uğraşmadan sadece @Bean ve @Configuration ile mesaj gönderip almanı sağlar.
- Üretici (Producer) ve tüketici (Consumer) tanımlarını basit Java koduyla yapabilmenizi sağlar.
- Mesajlaşma altyapısını (transport layer) soyutlar

Not: Spring Cloud Stream, bazı temel resiliency mekanizmalarını kendi içinde sağlar ama bunlar Resilience4j ile değil, **Kafka binder** veya **Spring Retry** aracılığıyla sağlanır:

Not: Circuit Breaker, Bulkhead, TimeLimiter gibi gelişmiş resiliency yoktur

Spring Cloud Stream Mimarisi

Özellik	Açıklama
Transport abstraction	Kafka, RabbitMQ gibi sistemlerden bağımsız kod yazarsın
Otomatik topic oluşturma	İster topic kendin açarsın, ister framework halleder
Binding configuration	YAML Üzerinden bağlama yapılır (bindings)
Stream processing support	Consumer, Function, Supplier desteği
Scalable consumer groups	Kafka'nın partition & group mantığıyla uyumlu

Bağımlilikler

Kafka Binder: Spring Cloud Stream'in Kafka ile konuşmasını sağlayan özel bir bağlayıcı (adapter) modülüdür.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

Not: Kafka üzerinde topic kullanımı yaparken
<functionName>-in|out-0 mantığı kullanılır.

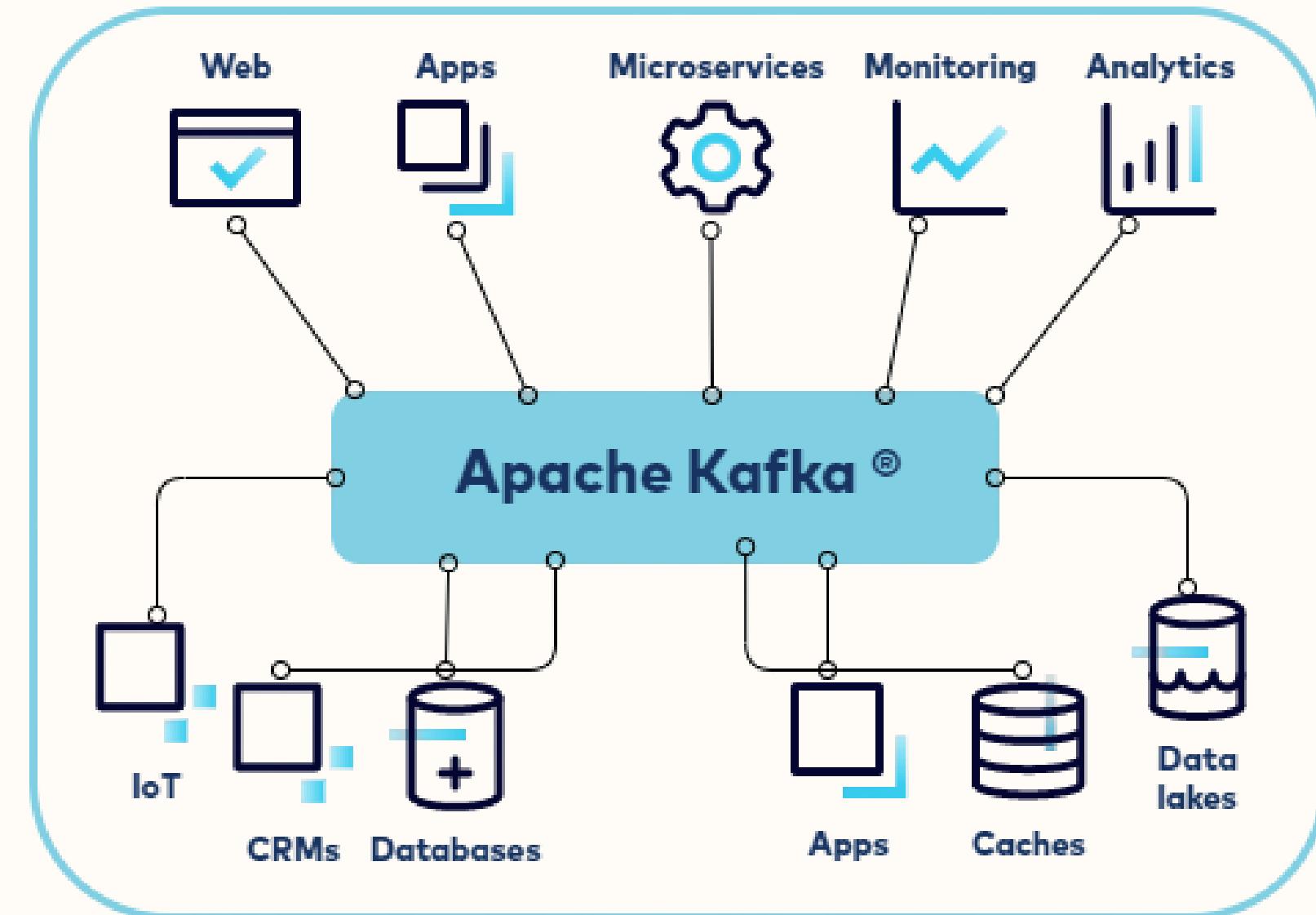
Not: Spring Cloud Stream tek başına sadece abstract transport layer sağlar. Binder olarak kafka veya rabbitmq gibi adapterlar eklenmelidir

Kafka Konfigürasyonu

```
cloud:  
stream:  
function:  
    autodetect: true  
bindings:  
    submitOrder-in-0:  
        destination: orders-topic  
        content-type: application/json  
    submitOrder-out-0:  
        destination: orders-topic  
        content-type: application/json  
kafka:  
    binder:  
        auto-create-topics: true  
        brokers: localhost:9092
```

Apache Kafka

Apache Kafka, LinkedIn tarafından geliştirilip daha sonra Apache Foundation'a bağışlanan, dağıtık, yüksek performanslı bir mesajlaşma ve veri akışı (streaming) platformudur.



Kafka'nın Ana Özellikleri

Dağıtık mimari → Veriler farklı sunuculara (broker) dağıtılr, böylece yüksek hacimli veriler işlenebilir.

Yüksek performanslı → Saniyede milyonlarca mesaj işleyebilir.

Dayanıklı ve güvenilir → Mesajlar disk üzerinde saklanır, kopyalanır (replication).

Gerçek zamanlı & batch işleme → Hem anlık hem de toplu veri akışı yapılabilir.

Hizmetler arası iletişim → Mikroservisler arasında veri alışverişini etkin kılar.

KAFKA Mimarisi

Topic

Topic, Kafka'da mesajların kategorize edildiği veya gruplandığı "kanal"dır.

Partition

Kafka'da mesajların fiziksel olarak sıralı şekilde saklandığı bölümdür. Paralel mesaj işleme ve ölçeklenebilirlik sağlar

Offset

Her mesajın partition içindeki sırasını belirleyen sayıya ise offset denir.

Producer

Kafka'ya veri (mesaj) gönderen uygulama veya bileşendir.

Consumer

Kafka'dan mesajları okuyan ve işleyen uygulama veya bileşendir.

Consumer Group

Birden fazla consumer'ı bir arada tutar, topic partition'larını paylaşarak mesajların dengeli ve tekrarsız tüketilmesini sağlar.

Producer

- Üreticiler (producers) mesajları belirli bir topic'e gönderir.
- Tüketiciler (consumers) ise bu topic'ten mesajları okur.
- Topic'ler, veri akışının mantıksal ayrimıdır; örneğin orders, payments gibi farklı iş alanlarına göre ayrılır.
- Her topic, bir veya daha fazla partition'a bölünerek paralel işleme ve ölçeklenebilirlik sağlar.

Consumer

- Kafka broker'dan mesajları offset sırasına göre okur.
- Her partition için okuduğu son offset'i takip eder.
- Mesajları iş mantığına göre işler.
- Consumer group içindeyse partitionları paylaşarak paralel çalışır.
- Partition içi mesaj sırasını korur.
- Mesaj işlendiğinde offset'i commit eder.
- Hata durumunda mesajı tekrar okuyabilir.

Topic

- Üreticiler (producers) mesajları belirli bir topic'e gönderir.
- Tüketiciler (consumers) ise bu topic'ten mesajları okur.
- Topic'ler, veri akışının mantıksal ayrimıdır; örneğin orders, payments gibi farklı iş alanlarına göre ayrılır.
- Her topic, bir veya daha fazla partition'a bölünerek paralel işleme ve ölçülebilirlik sağlar.

Not: Her event tipi kendi topic'inde üretilip tüketilmeldir. Bu sayede kafka partition ve offset yönetimi daha anlamlı olur, izleme, hata ayıklama ve ölçülebilirlik kolaylaşır.

Partition

Kafka'da mesajların fiziksel olarak sıralı şekilde saklandığı bölümdür.

- Her mesajın partition içindeki sırasını belirleyen sayıya ise **offset** denir.
- Offset sadece partition içinde anlamlıdır.
- Offset 0 olan bir mesaj partition-0'da başka bir mesaj, partition-1'de başka bir mesaj olabilir.
- Consumer, verileri Kafka'da her partition'dan sırayla, offset numaralarına göre okur.

Consumer Group

Birden fazla consumer'ı bir arada tutar, topic partition'larını paylaşarak mesajların dengeli ve tekrarsız tüketilmesini sağlar.

Birden Fazla Consumer'ı Birlikte Yönetir

- a. Aynı grup kimliğiyle (group.id) çalışan consumer'lar bir arada gruplanır.

Partitionları Paylaştırır (Load Balancing)

- a. Consumer group içindeki her consumer, topic'in partition'larını bölüşür.
- b. Böylece paralel ve dengeli tüketim sağlanır.

Consumer Group

Mesajları Tekrar Etmeden Okur

- a. Aynı grup içindeki consumer'lar, partition'lar arasında mesajları bölüşüp, aynı mesajı sadece bir consumer okur.

Offset Yönetimini Grup Bazında Yapar

- a. Offset bilgisi, consumer group'a ait olarak Kafka'da tutulur.
- b. Consumer group içindeki tüm tüketiciler, hangi mesajların okunduğunu birlikte takip eder.

Yüksek Ölçeklenebilirlik ve Fault Tolerance Sağlar

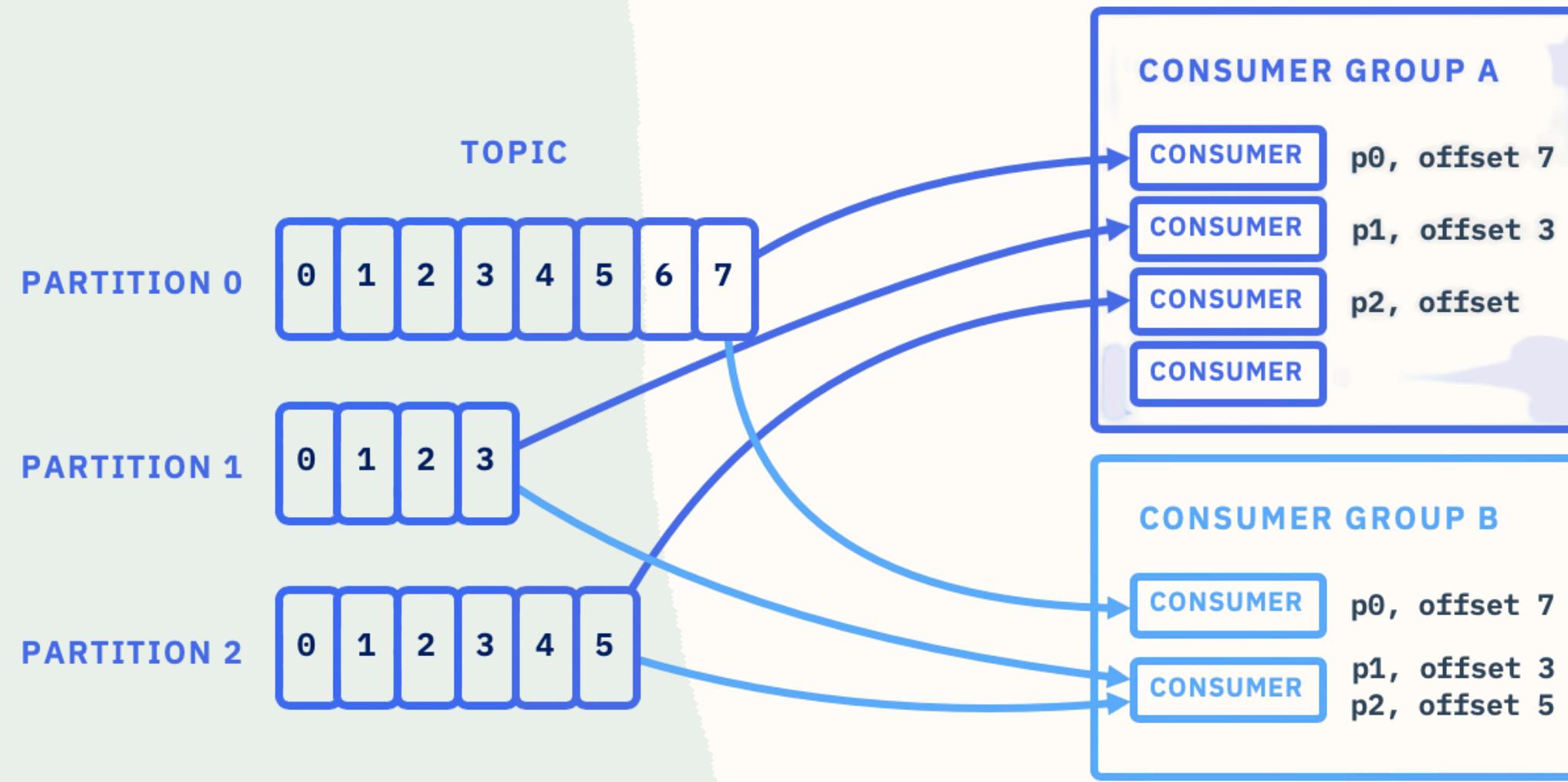
- a. Consumer sayısı arttıkça tüketim hızı artar.
- b. Consumer'lardan biri kapanırsa, kalanlar partitionları devralır.

Önemli Notlar !!!

Birden fazla topic'i aynı uygulama veya servis tüketiyorsa, bunlar genelde aynı group.id kullanır.

Farklı uygulamalarda aynı group.id ile çalışırsak, Partition'lar bu farklı uygulama instanceları arasında paylaştırılır. Bu, genelde istenmeyen bir durumdur, çünkü uygulamalar birbirinin mesajını "yarı yolda kapabilir"

Kafka Mimarisi

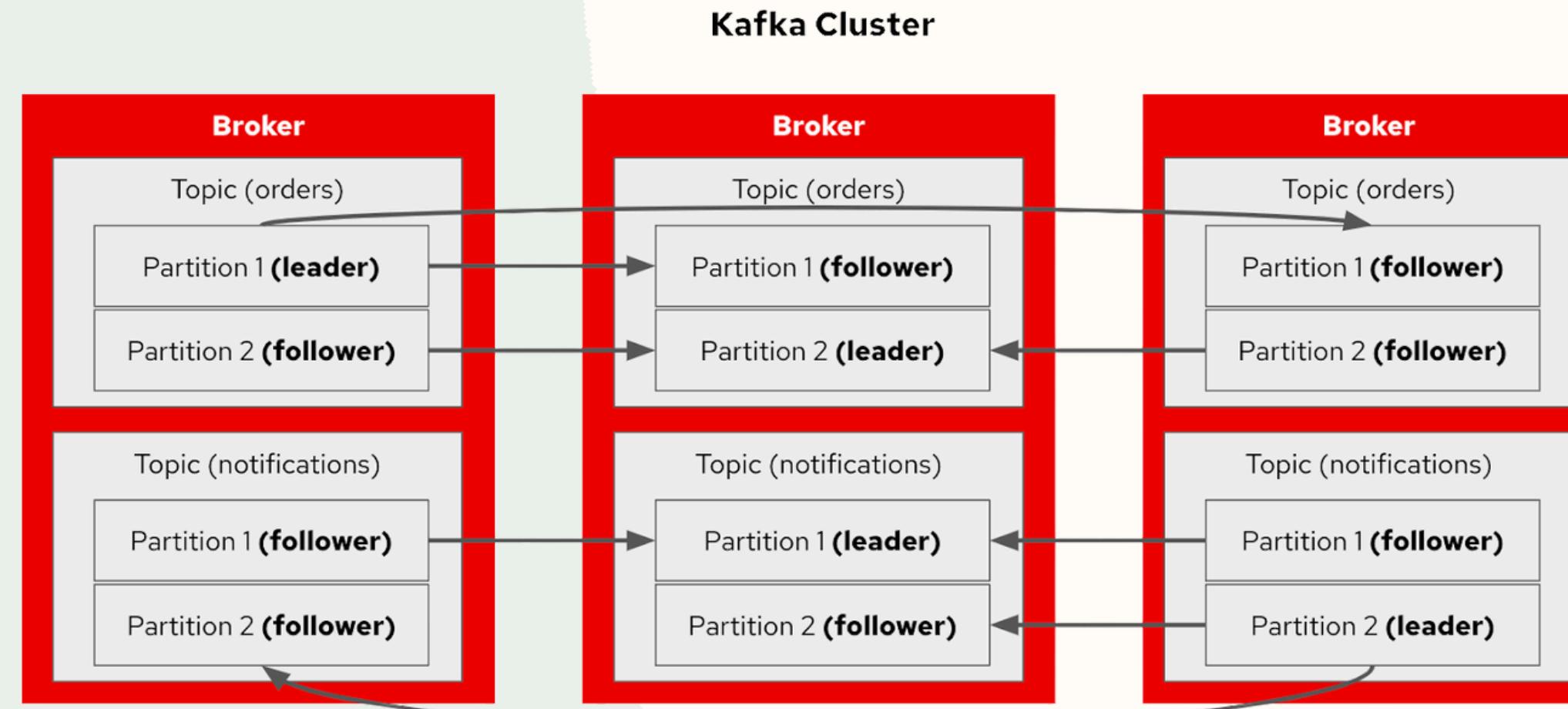


Bir topic'i farklı consumer group'lar dinliyorsa → Topic'in tüm partition'ları her gruba ayrı ayrı gönderilir.

Her partition, bir consumer group içinde yalnızca 1 consumer tarafından okunur.

Consumer group (group.id) → Kafka, bu grup içindeki tüketiciler arasında partition'ları paylaştırır.

Kafka Mimarisi



- Tüm okuma ve yazma işlemleri önce leader üzerinden yapılır.
- Consumer ve Producer talepleri Leader üzerinden karşılanır.
- Followers ise Leader'dan veriyi çekerek (replicate) kendi diskinde tutar
- Followers, Okuma ve yazma yapmaz, sadece leader'ı yedekler.
- Followers, Partition'ın replica'larıdır (kopiyaları).

Producer Idempotence

Idempotence, bir işlemin kaç kez tekrarlanırsa tekrarlansın aynı sonucu üretmesi özelliğidir.

- **Producer ID (PID)** → Kafka her producer'a benzersiz bir ID atar.
- **Sequence Number** → Producer her mesajına artan bir sıra numarası ekler.
- Kafka broker, bu numaraları takip ederek duplicate mesajları reddeder.

Kafka 0.11 – 2.3 arası → enable.idempotence default = false

Kafka 3.0 ve sonrası (Ekim 2021'den itibaren) → enable.idempotence default = true

Not: acks=all (broker onayı) zorunlu hale gelir.

Not: enable.idempotence: true; aynı mesajın broker'da birden fazla kaydedilmesini engellemek

Producer Acknowledgement

Kafka producer, bir mesajı broker'a gönderdiğinde broker'ın mesajı onaylama davranışını kontrol eden ayardır.

acks : mesaj güvenliği seviyesi

acks 0: Producer ack beklemez → maksimum hız ama veri kaybı riski yüksek

acks 1: Leader broker mesajı aldı → başarılı sayılır, follower'lar beklenmez

acks all: Leader + tüm in-sync replica'lar mesajı aldı → maksimum güvenlik (en güvenli)

KAFKA ISR (In-Sync Replicas):

Leader ile güncel (senkron) olan follower replikalarının listesidir.

Bu liste, Kafka tarafından otomatik olarak yönetilir, manuel yapılandırılmaz.

Eğer acks=all olarak ayarlanırsa, Kafka mesajı tüm ISR üyelerine başarıyla ulaştıktan sonra başarılı sayar.

Yüksek erişilebilirlik, veri tutarlılığı ve güvenli lider değişimi sağlar.

Eğer leader broker down olursa, ISR'deki follower'lardan biri yeni leader olur

Not: Kafka'da bir follower, belirli bir süre (örneğin 10sn) boyunca leader'dan veri alamazsa, ISR'den çıkarılır. Eğer tekrar senkron hale gelirse, ISR'ye otomatik olarak geri alınır

KAFKA Retention Policy

Kafka'da mesajlar retention policy'ye göre silinir

Zaman Bazlı (time-based retention) (default)

a.log.retention.hours=168 # 7 gün

Boyut Bazlı (size-based retention)

a.log.retention.bytes=1073741824 # 1 GB

Log Compaction (sadece en güncel olanı sakla)

a.cleanup.policy=compact

KAFKA Dead Letter Queue

Bir mesajın normal işleme (consume) sırasında hata vermesi veya işlenememesi durumunda, o mesajın kaybolmaması için gönderildiği özel bir kuyruktur.

Böylece, problemli mesajlar normal kuyruğun dışına çıkarılır ve sistemin genel işleyişi aksatılmadan devam eder.

Not: DLQ'de toplanan mesajlar, hata analizi, sorun giderme veya yeniden işleme (reprocessing) için daha sonra ayrı olarak incelenir.

Neden Dead Letter Queue kullanılır?

- Mesaj işlenirken hatalar çıkabilir (örneğin, veri formatı bozuk, geçersiz veri, geçici servis kesintisi).
- Bu tür hatalı mesajlar sürekli yeniden işlenmeye çalışılırsa, sistem tıkanabilir veya diğer mesajların işlenmesi gecikir.
- DLQ, bu mesajları ayırarak sistemi korur.
- İşlemesi mümkün olmayan mesajlar burada toplanır ve geliştiriciler veya operasyon ekibi tarafından incelenip müdahale edilir.

Mikro Hizmet Mimarilerde Veritabanı Tasarımı

Database per Service

- Her mikroservis kendi veritabanına sahip olmalı.
- Servisler arası doğrudan DB erişimi yasak olmalı, API veya event üzerinden haberleşmeli.

Servis Sınırları ve Bounded Context

- Domain-driven design (DDD) kullan: her servis kendi **bounded context** içindeki veri modelini yönetir.

Asenkron Veri Paylaşımı

- Servisler arası veri paylaşımı event-driven olmalı (Kafka, RabbitMQ).
- **Eventual consistency** kabul edilmeli, **strong consistency** çoğu zaman zor ve maliyetlidir

Mikro Hizmet Mimarilerde Veritabanı Tasarımı

Saga ve Transaction Yönetimi

- Çoklu servis transaction'ları için saga pattern veya outbox pattern kullan.
- ACID yerine eventual consistency modelini benimse.

Polyglot Persistence

- Servis ihtiyacına göre farklı DB tipi seçilebilir:
 - Relational (Postgres, MySQL)
 - NoSQL (MongoDB, Cassandra)
 - Stream / Time-series (Kafka Streams, InfluxDB)

Mikro Hizmet Mimarilerde Veritabanı Tasarımı

Avoid Cross-Service Joins

- Servisler arası join işlemleri yapılmamalı → performans ve coupling problemi olur.
- Gerekirse denormalize veya cache kullan.
-

Independent Scaling

- Her veritabanı servis bazında ölçeklenebilir olmalı → DB replication veya sharding ile.
-

Schema Evolution & Versioning

- Mikroservis DB'sinde şema değişiklikleri için versioning uygulayın.
- Kod tabanlı Migration ile veritabanı değişiklikleri yapmaya çalışın.

Mikroservislerde Denormalizasyon

- Mikroservisler loosely coupled olmalı → servisler arası join yapmak önerilmez
- Performans artırmak için read model hızlı olmalı
- Eventual consistency kabul edildiği için denormalize veriyi güncellemek sorun değil

Product Catalog Micro Service

Product.cs

ProductName
ProductPrice
ProductStock

Order Micro Service

Order.cs

OrderNumber
CustomerName
ShipAddress
ShippedDate
OrderItems

OrderItems.cs

ProductName
ProductPrice
Quantity

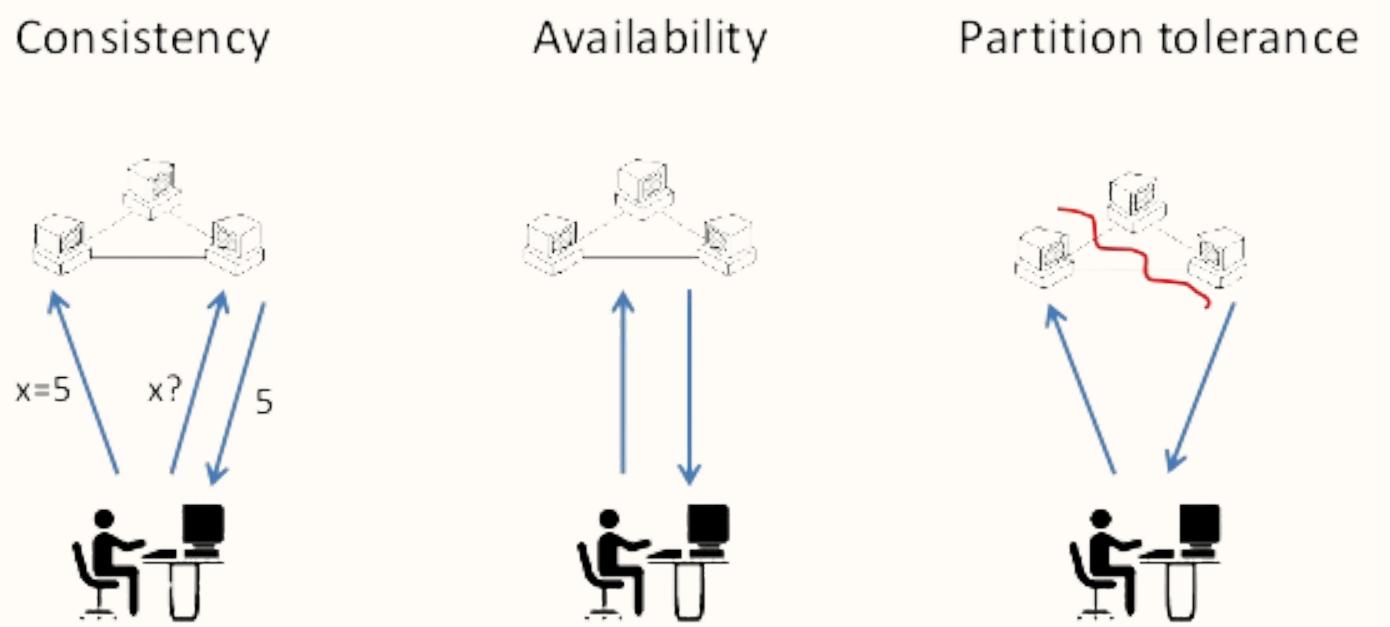
CAP Teoremi

CAP Teoremi → bir dağıtık sistem aynı anda C, A, P üçlüsünü tam sağlayamaz

Consistency (Tutarlılık): Tüm node'lar aynı anda aynı veriyi görür

Availability (Erişilebilirlik): Her isteğe her zaman cevap verir

Partition Tolerance (Bölünme Toleransı) : Network partition veya node failure durumunda çalışmaya devam eder



CAP Teoremi

- Mikroservisler distributed system olduğundan CAP geçerlidir
- Mikroservis tasarımda çoğu zaman P + C veya P + A seçilir
- Bu seçim domain ve iş gereksinimlerine göre yapılır

Örn:

CP: Tutarlı ama bazı zamanlar erişilemez (örn. banka transferi)

AP: Her zaman erişilebilir ama kısa süreli veri tutarsızlığı olabilir (örn. sosyal medya feed)

Distributed Transaction Yönetimi

3 farklı yöntem kullanabiliriz.

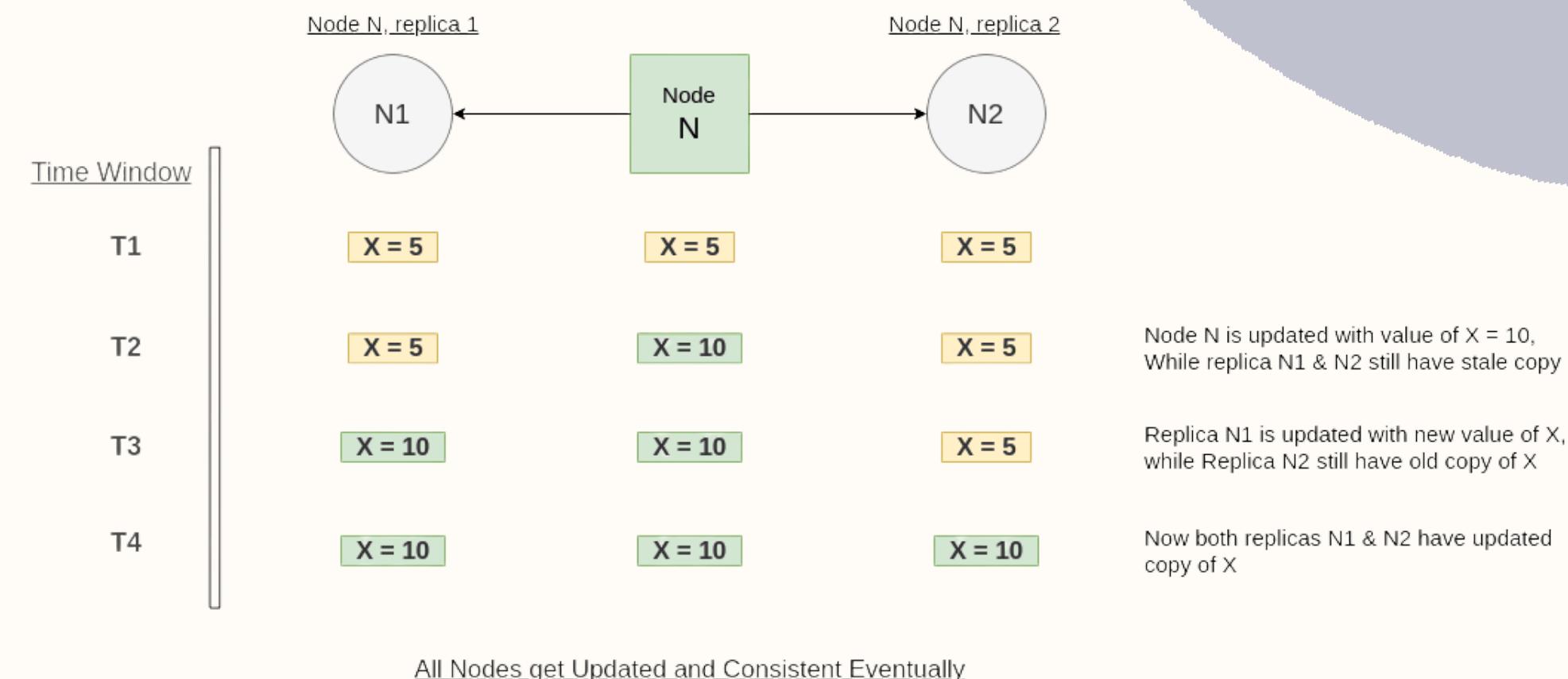
- Eventual Consistency
- SAGA Pattern
- Outbox Pattern

Eventual Consistency

Distributed transaction ve mikroservis mimarisinde bir tutarlılık yaklaşımıdır.

“Tüm sistem bir süre sonra tutarlı duruma gelir.”

Yani, anlık tutarlılık (strong consistency) garanti edilmez, ama zaman içinde tüm kopyalar eşleşir.



SAGA PATTERN

Mikroservislerde distributed transaction yönetme yöntemidir

Çoklu servislerdeki işlemlerin tutarlı bir şekilde tamamlanmasını sağlar.

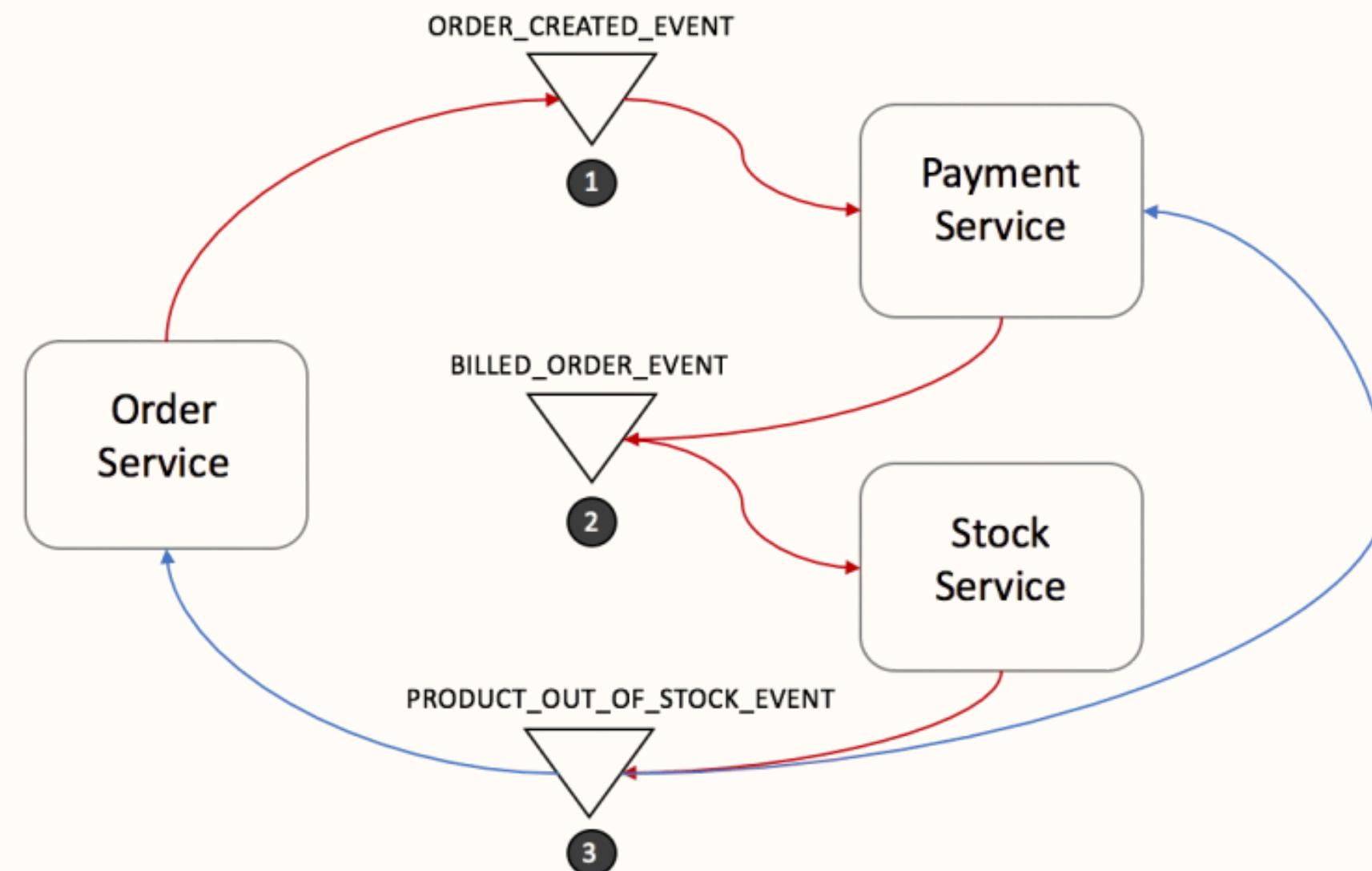
Klasik ACID transaction yerine her adım kendi **local transaction'ını** yapar ve başarısız olursa **compensating** transaction ile geri alınır.

Choreography: Servisler event-driven birbirini tetikler, merkezi koordinatör yok

Orchestration: Merkezi saga orchestrator yönetir, hangi servisin ne zaman çalışacağını bilir

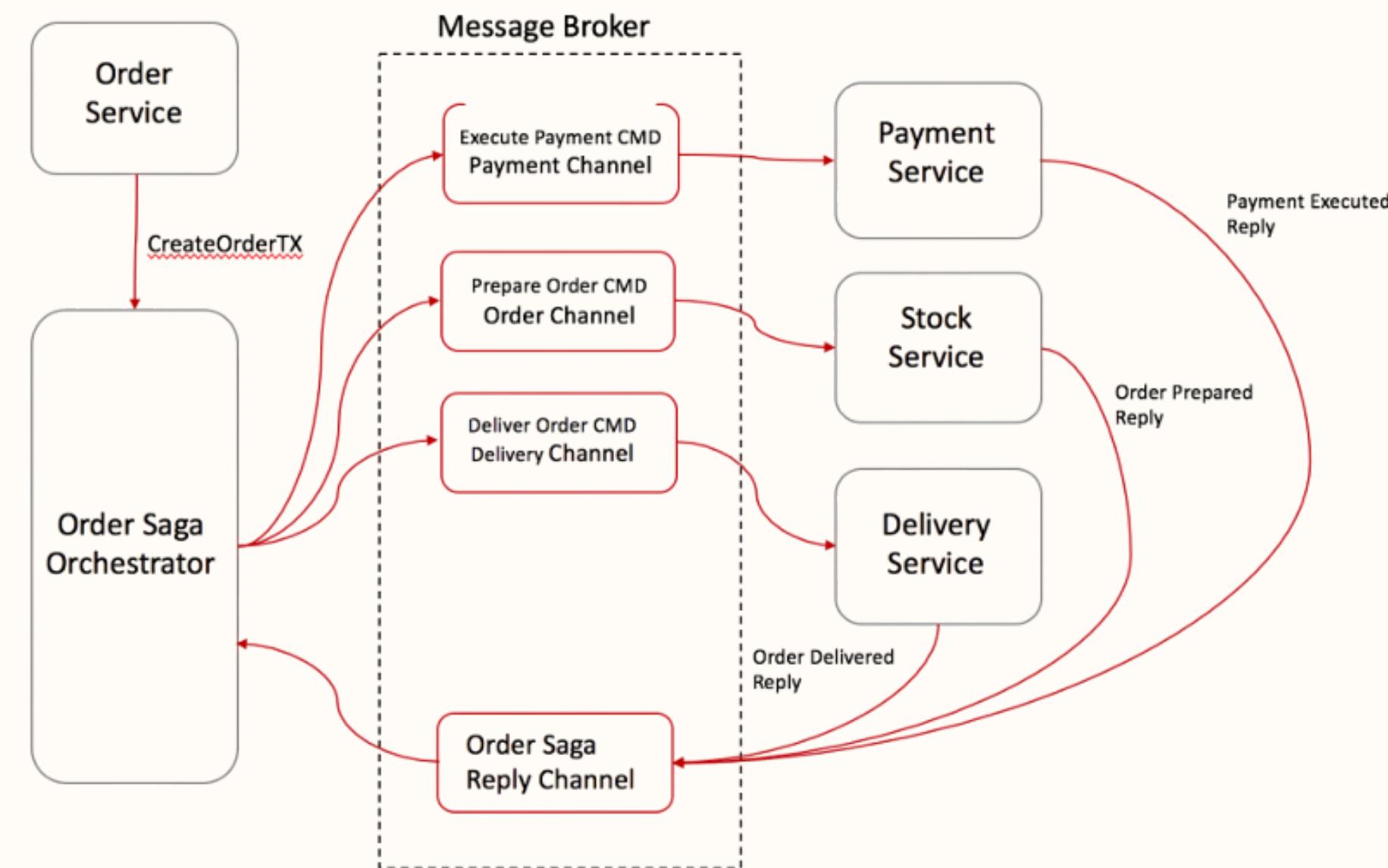
Choreography Saga

Bu yaklaşımın merkezi bir yönetici yoktur. Her servis işlemi tamamlar ve event fırlatır ve bir sonraki servis ilgili event consume edip sürecine devam eder.



Orchestration Saga

Bu yaklaşımda tüm işlemleri yöneten bir Saga orchestrator vardır. Bu orchestrator subscribe olan tüm consumer servislere ne zaman ne yapacağını iletken bir yapıdadır.



Orchestration

- Merkezi koordinasyon, kolay yönetim
- Rollback / Compensating Transaction Merkezi
- Orchestrator hangi adımın ne zaman çalışacağını bilir
- Servislerin Orchestrator bağımlılığı var
- Bottleneck oluşma ihtimali

Choreography

- Her servis kendi başına event publish/subscribe ile hareket eder
- Servisler kendi rollback işlemini yapar, koordinasyon zor
- Merkezi bağımlılık yok, servisler loosely coupled
- İzlemek zor, event log'lara bakmak gereklidir

Outbox Pattern

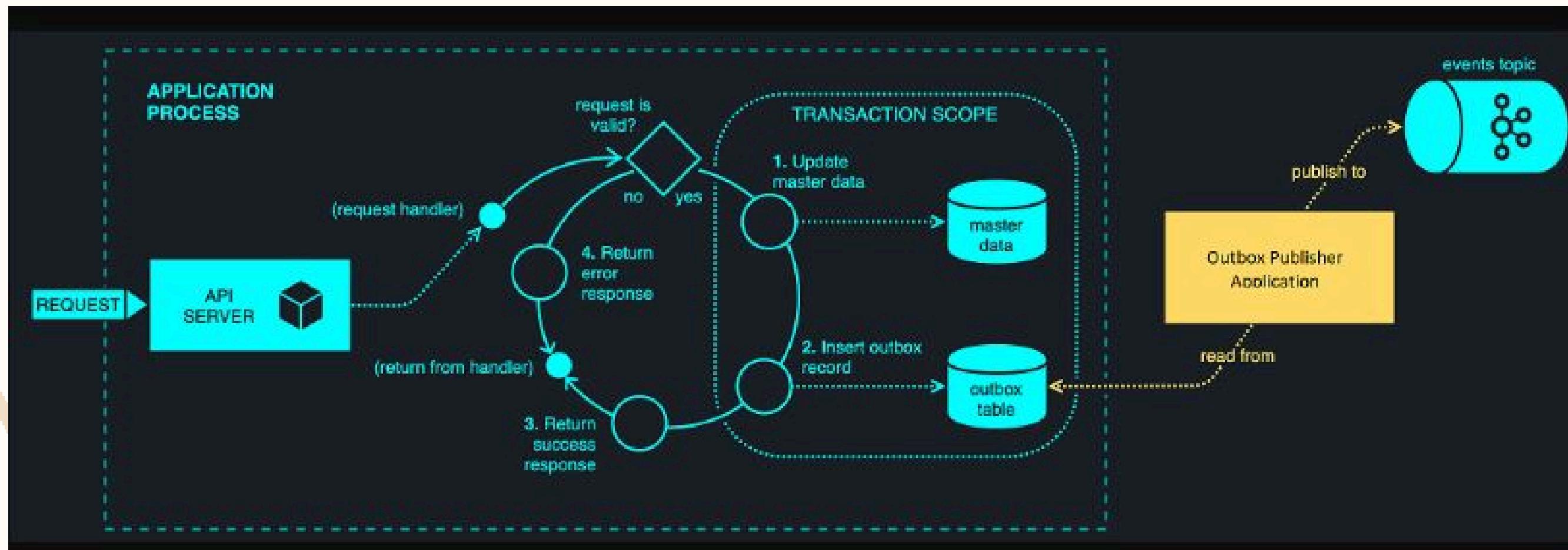
Mikroservislerde veritabanı değişiklikleri ile event yayınıni atomik yapmak ve mesaj kaybını önlemek.

Yani: “DB transaction tamamlanmadan event publish edilmesin” problemi çözülür.

Nasıl Uygulanır ?

- Servis kendi DB transaction’ı içinde hem veriyi değiştirir hem de outbox tablosuna event yazar. Bu işleme **dual write** işlemi denir.
- Daha sonra ayrı bir process veya poller, outbox tablosundan eventleri alır ve message broker'a (Kafka, RabbitMQ) publish eder

Outbox Pattern



- Atomicity: Veri değişikliği ve event yazma tek transaction'da
- Mesaj kaybı yok: Event publish hatalarında tekrar denenebilir
- Eventual consistency sağlar

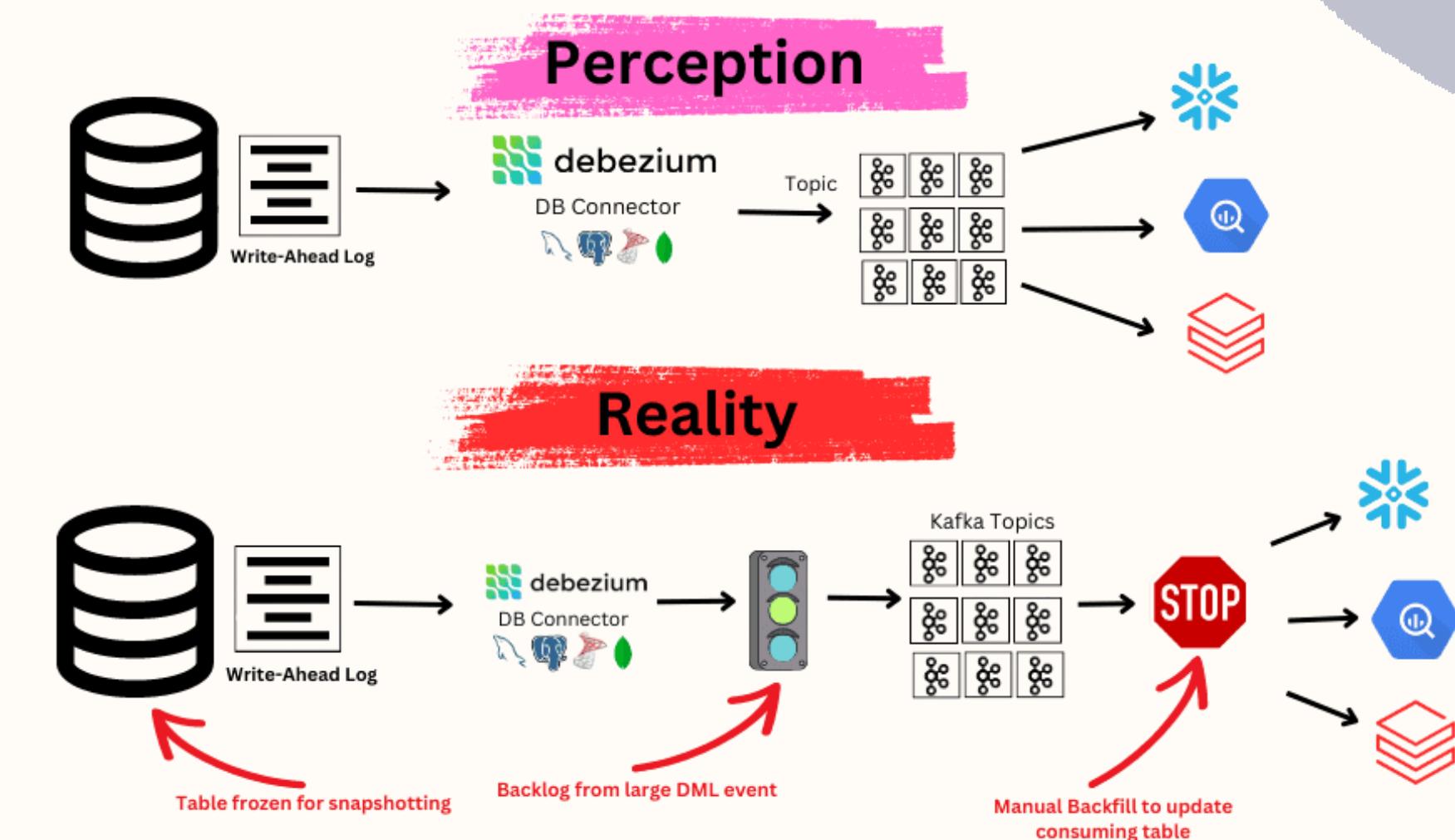
Debezium

Debezium, açık kaynaklı bir **Change Data Capture (CDC)** platformudur.

Amaç: Veritabanındaki değişiklikleri gerçek zamanlı olarak event stream'e dönüştürmek.

Desteklediği veritabanları:

- PostgreSQL,
- MySQL,
- SQL Server,
- MongoDB,
- Oracle



Debezium

Debezium'un Çalışma Mantığı

1. Debezium, veritabanı loglarını (**binlog** / **WAL** / **redo log**) okur.
2. Her değişikliği (insert, update, delete) bir event olarak üretir.
3. Bu event'i Kafka topic'ine veya başka bir event bus'a yollar.
4. Mikroservisler bu event'leri subscribe ederek kendi DB veya iş mantığını günceller.

Mikroservislerde distributed transaction yerine Outbox Pattern + Debezium sık kullanılır.

Bağımlılıklar

Kafka ve Zookeeper kurulu olmalı

Debezium connector (örn. PostgreSQL connector)

Kafka consumer olarak mikroservisler

Avantajları

Atomicity: Outbox + DB transaction tek atomic operasyon

Mesaj kaybı yok: Debezium, veritabanı loglarını okuduğu için hiçbir event kaçmaz

Loose Coupling: Servisler doğrudan DB'ye erişmez

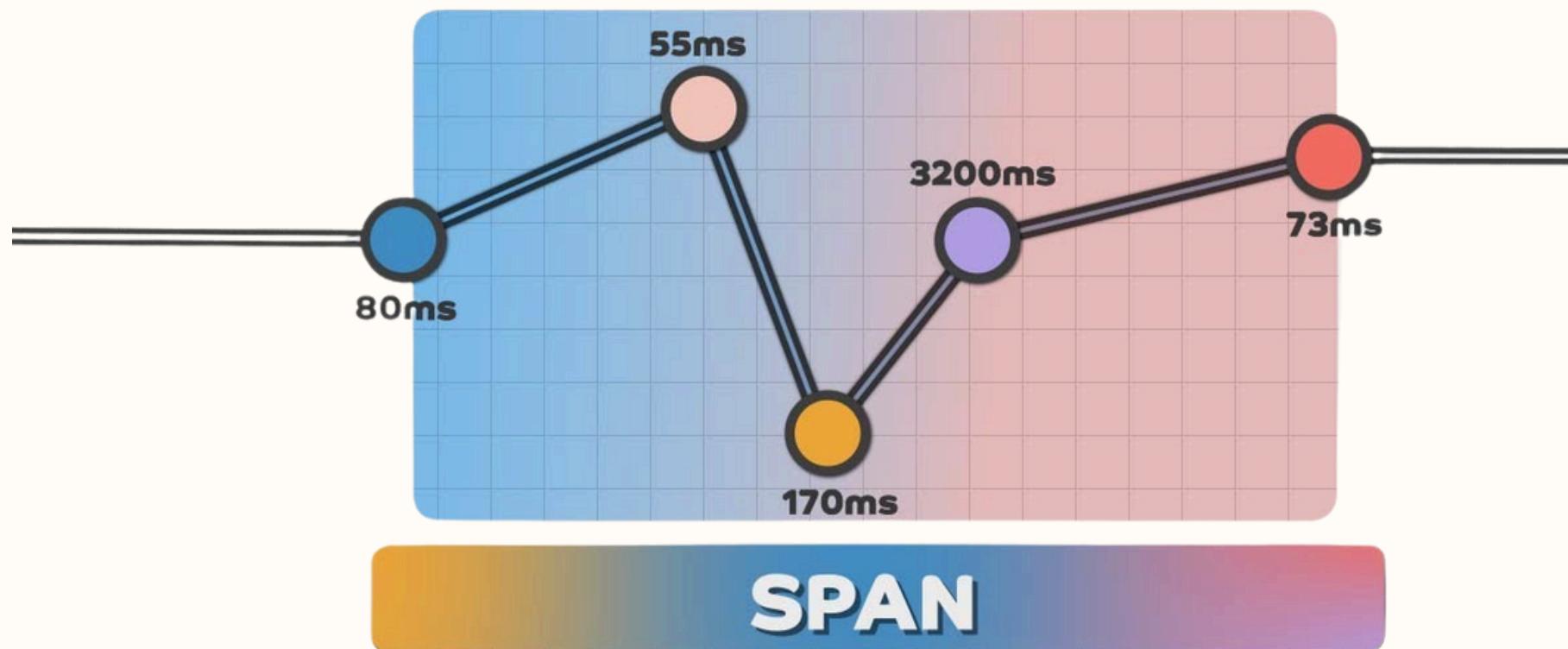
Eventual Consistency: Tüm mikroservisler eventleri alıp state'lerini günceller

Distributed Tracing

Mikro hizmet mimarilerinde bir isteğin sistem genelindeki tüm yolculuğunu takip etmeye yarayan bir tekniktir.

Mikro hizmetlerde bir istek, birden fazla mikroservisten geçerek sonuçlanabilir.

Bu zincir kırıldığında hangi servis sorumlu, nerede ne kadar süre harcandı, hangi adım hata verdi gibi soruların yanıtı kolay değildir.



Distributed Tracing Kavramı

Bir kullanıcı isteğiinin sistemdeki tüm yolculuğunu temsil eder.

Kullanıcı Sipariş Süreci : Gateway → Auth → Order → Payment

- Her dış (kullanıcıdan gelen) istek için benzersiz bir Trace ID oluşturulur.
- Bu Trace ID, mikro hizmetler arasında taşınır (HTTP header, mesaj kuyruğu metadata vb. yoluyla).
- Böylece, istek başlangıçtan sonuna kadar aynı Trace ID ile izlenir, hangi mikro hizmetten geçerse geçsin bu ID korunur.
- Mikro hizmetler içinde bu trace ID altında farklı işlemler Span oluşturulur.

Span Kavramı

Bir trace içindeki bireysel işlemleri temsil eder.

Kullanıcı Sipariş Süreci : Gateway → Auth → Order → Payment

- Gateway → Span1
 - Auth → Span2
 - Order → Span3
 - Payment → Span4

Not: Spanler arasında Parent → Child ilişkisi vardır ve Aynı istekdeki tüm spanlar bir trace'da bağlıdır.

Not: Her Span, Başlangıç/bitiş zamanı, Adı (operation name), Hata durumu, Etiketler (tags), Event'ler içerir.

Micrometer

JVM tabanlı uygulamalarda ölçülebilir metrikler (metrics) toplamak ve bunları dış sistemlere göndermek için kullanılan, Spring Boot ile entegre çalışan bir kütüphanedir.

Ne tür veriler toplar?

- http.server.requests, response süresi, status kodları
- Heap kullanımı, GC, thread sayısı
- CPU kullanımı, disk I/O
- Aktif bağlantılar, bekleyen işlemler
- Uygulamaya özel sayaçlar, zamanlayıcılar (Custom)
- Verileri **Prometheus**, **Elastic** gibi metric görselleştirme toollarına iletir.

Not: Micrometer, metrics (ölçüm) içindir. Tracing (izleme) yapmaz.

Spring Boot Actuator

Spring Boot uygulamalarına hazır izleme, sağlık, metrik, bilgi (info), log seviyesi değiştirme gibi endpoint'ler ekler.

Bu endpoint'ler aracılığıyla uygulama durumu ve performansına dair bilgiler sunar.

- **Spring Boot Actuator**, metrics toplama işini **Micrometer**'a devreder.
- **Micrometer**, JVM, HTTP, DB, Thread Pool vb. metrikleri toplar ve bunları actuator endpoint'lerinden (`/actuator/metrics`) erişilebilir hale getirir.
- **Spring Boot Admin**, Actuator endpoint'lerini düzenli olarak sorgular ve metrikleri kendi arayüzünde gösterir.

Brave Nedir?

Brave, Java için geliştirilmiş bir instrumentation (izleme) kütüphanesidir. OpenZipkin projesinin bir parçasıdır.

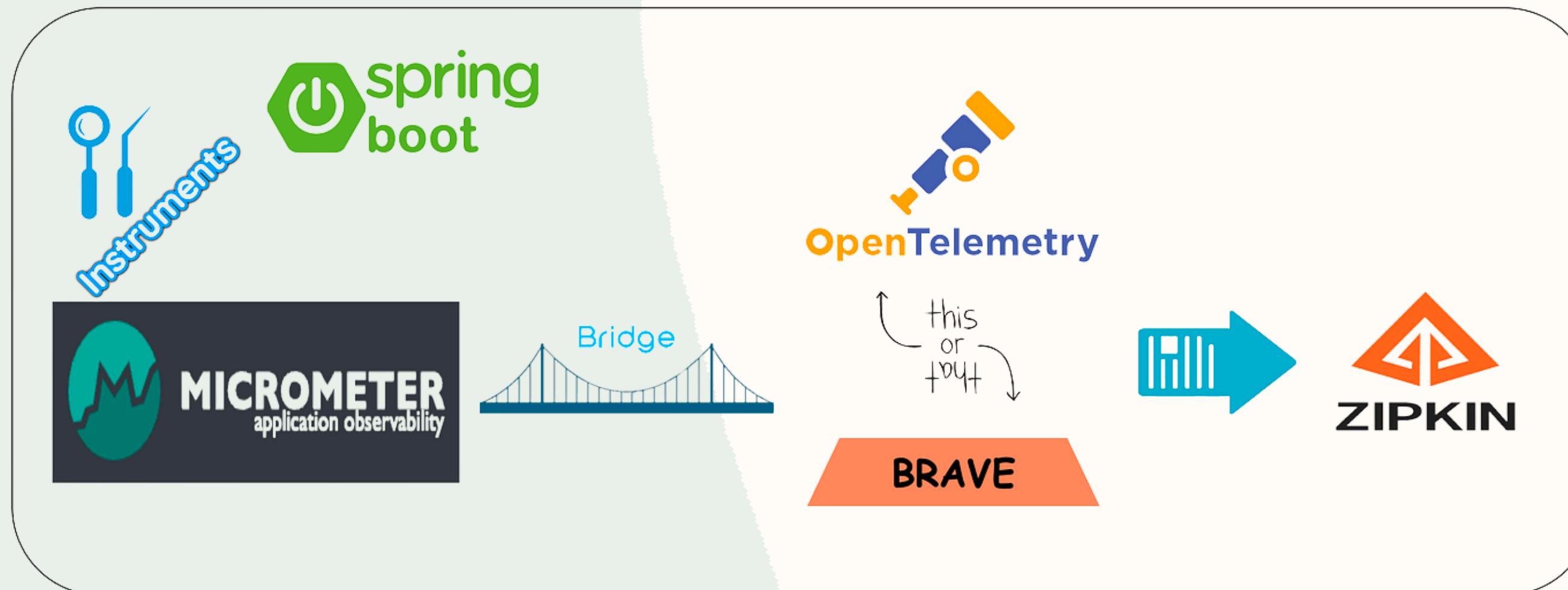
Ne işe yarar?

- Uygulamalara trace ve span üretme yeteneği kazandırır.
- Spring Boot ile entegre çalışır.
- Gelen HTTP isteklerinden Trace ID'yi alır (ya da üretir), outgoing (çıkış) isteklerde bu ID'yi taşır.
- Spans'leri Zipkin gibi sistemlere gönderir.

Özellikleri:

- HTTP Client/Server interceptors
- Spring MVC, WebFlux, gRPC destekleri
- Asenkron çağrıları izleyebilme yeteneği
- Custom span oluşturma

Spring Cloud Projelerinde Distributed Tracing



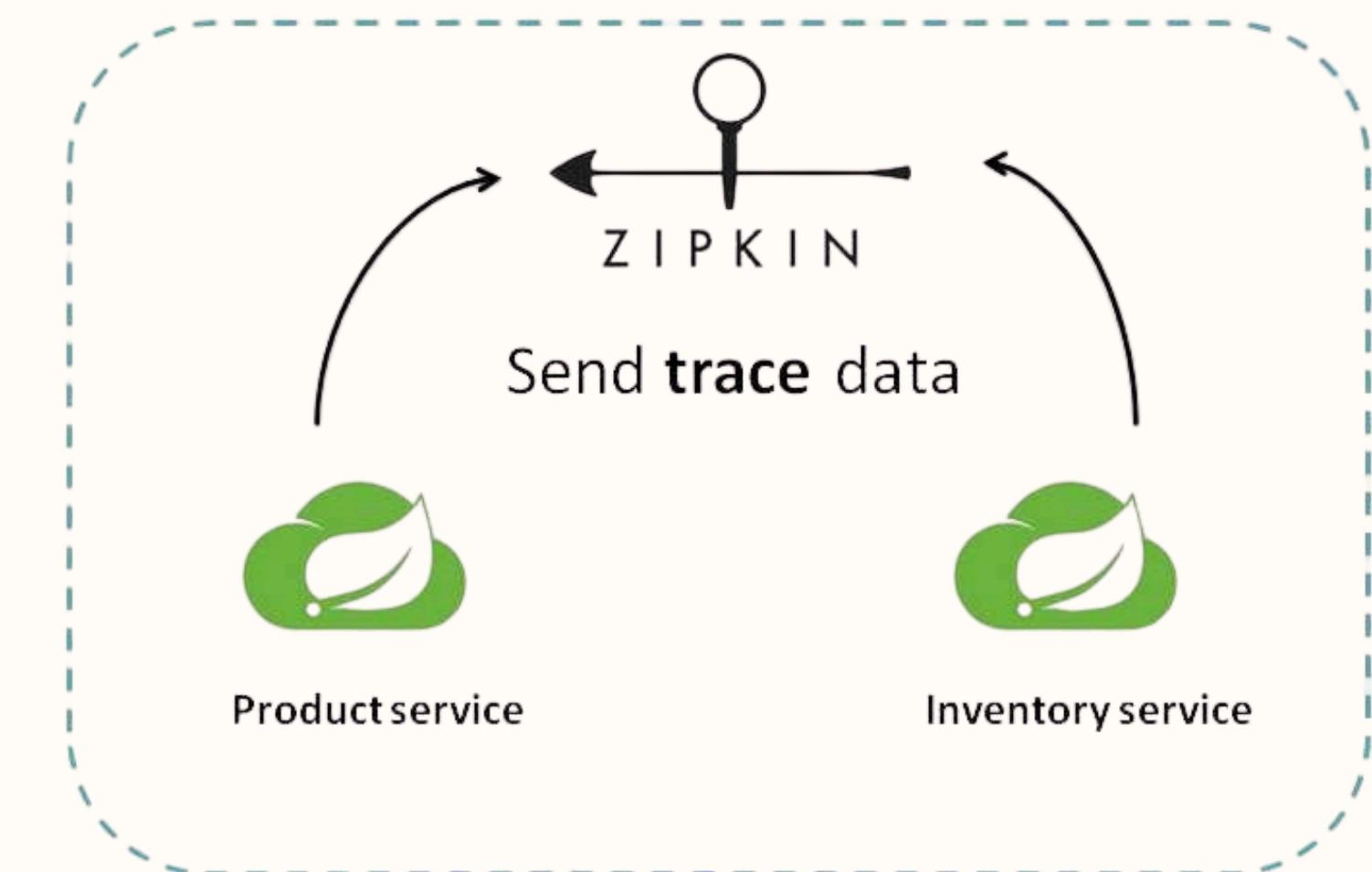
Zipkin Tracing Tool

Açık kaynaklı, dağıtık izleme (distributed tracing) sistemi için kullanılan bir araçtır.

Mikro hizmet mimarilerinde, gelen bir isteğin farklı servisler arasındaki yolculüğünü (trace) toplar, depolar ve görselleştirir.

Ne yapar ?

- Mikro hizmetler arasındaki karmaşık çağrı zincirini uçtan uca takip eder
- Hangi servisin ne kadar sürede çalıştığını gösterir
- Problemlı noktaları ve hataları tespit etmeyi kolaylaştırır



Zipkin Mimarisi

- **Collector:** Trace verilerini toplar ve saklar
- **Storage:** Trace verilerinin kalıcı olarak depolanması (örn: MySQL, Cassandra, Elasticsearch)
- **Query:** Depolanan verilerin sorgulanması
- **Dahboard:** Trace ve span'lerin görselleştirilmesi

Zipkin'in backend'i genellikle bir REST API olarak çalışır; trace verilerini HTTP POST ile alır.

Zipkin Mimarisi

Kullanım Alanları;

- Performans sorunlarını tespit etmek
- Servisler arası gecikmeleri analiz etmek
- Hangi servislerin hataya sebep olduğunu bulmak
- Uçtan uca istek takibi ve görselleştirme

Bağımlilikler

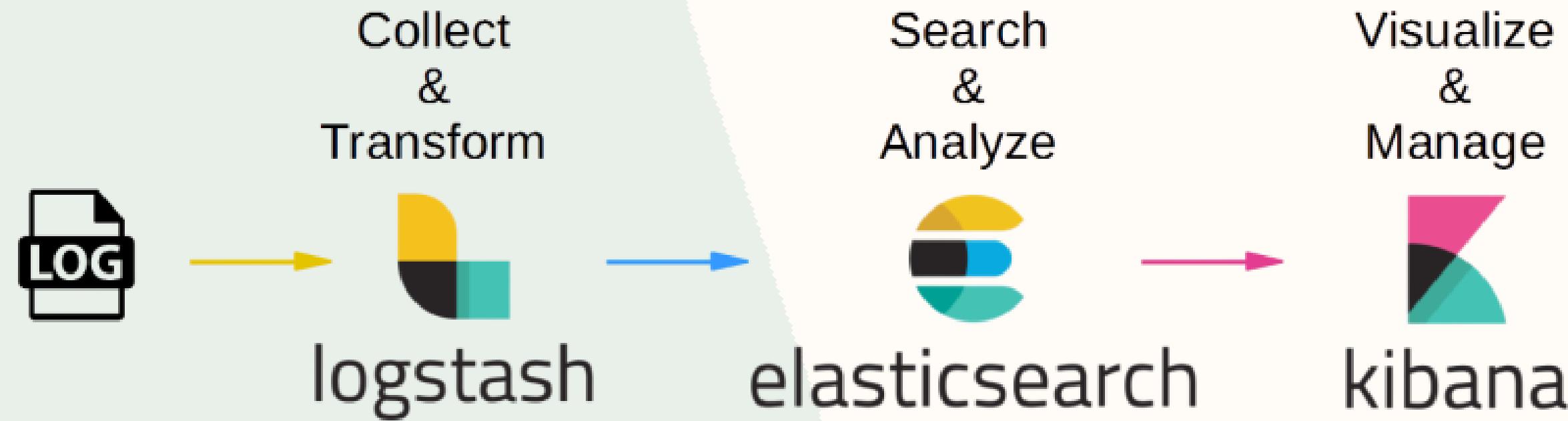
```
org.springframework.boot.spring-boot-starter-actuator  
io.micrometer.micrometer-tracing  
io.micrometer.micrometer-tracing-bridge-brave  
io.zipkin.reporter2.zipkin-reporter-brave.3.3.0  
io.zipkin.reporter2.zipkin-sender-okhttp3.3.3.0
```

Not: Tracing otomatik olarak oluşması için Spring Boot projenizde Micrometer Tracing + Actuator + Distributed Tracing starter'ları (Spring Cloud Sleuth veya benzeri bir yapı) olmalıdır.

Distributed Logging

Distributed Logging (Dağıtık Loglama), mikro hizmet mimarilerinde farklı servislerde üretilen logların merkezi bir yerde toplanması ve yönetilmesi sürecidir.

- Böylece tek bir noktadan tüm servislerin logları izlenebilir.
- Tek bir isteğin işlenmesi, birden çok servisten geçebilir. Bu süreci takip etmek için her servisin loguna bakmak gereklidir.



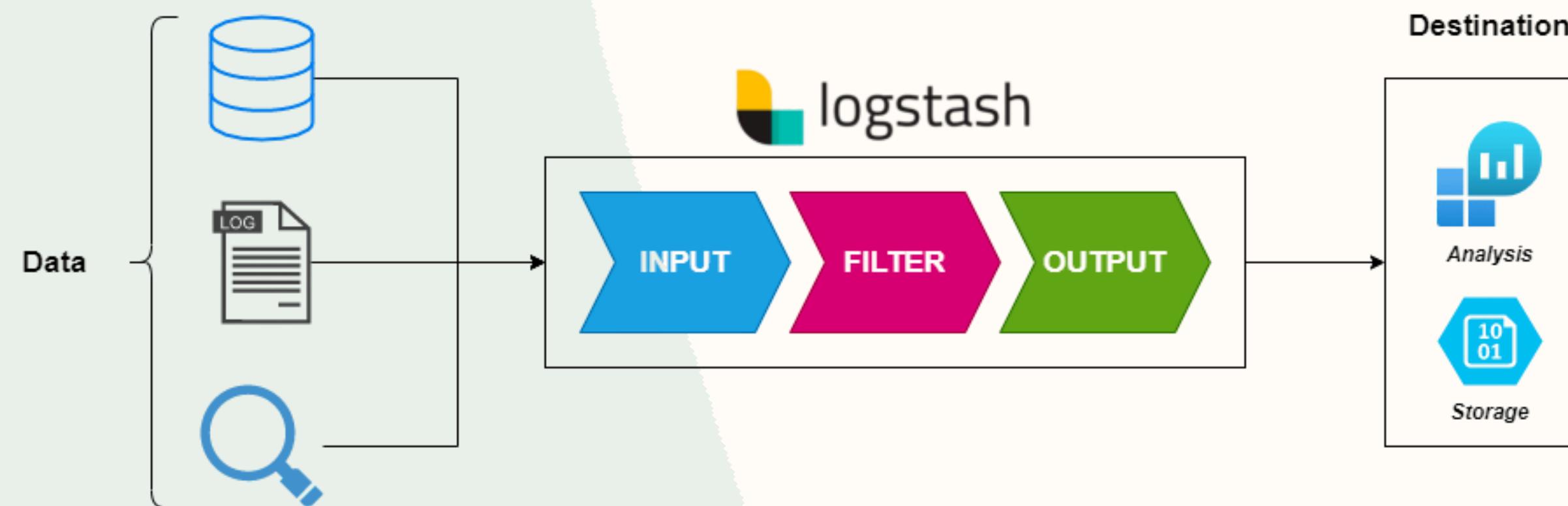
Distributed Logging

Genellikle şu mimari kullanılır:

1. Servisler log üretir (Logback, Log4j, vs.)
2. Loglar dosyaya veya TCP/UDP soketine yazılır.
3. Log toplayıcı bir araç (Filebeat, Fluentd, Logstash...) bu logları alır.
4. Merkezi sistem (Elasticsearch gibi) logları saklar ve indeksler.
5. Görüntüleme aracı (Kibana, Grafana) ile loglar analiz edilir.

Logstash

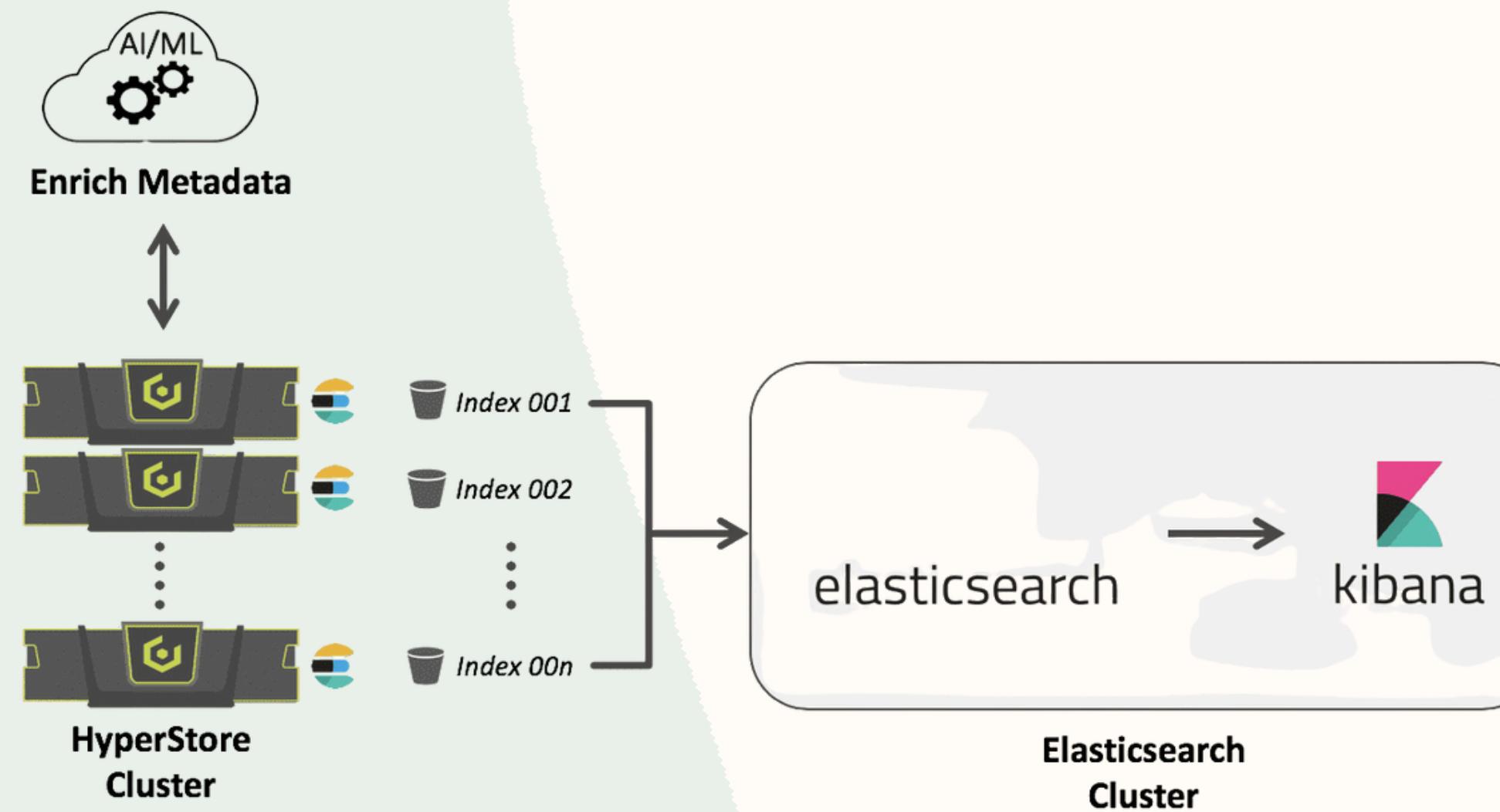
Çeşitli kaynaklardan gelen logları toplar, işler (dönüştürür) ve başka bir sisteme gönderen bir veri işleme aracıdır.



Elasticsearch

arama ve analiz için kullanılan güçlü bir arama motorudur.

Log gibi büyük verileri, İndeksleyerek hızlı erişim sağlar, Filtreleme, sorgulama, gruplama ve analiz imkanı verir.



Kibana

Kibana, Elasticsearch'teki verileri görsel olarak gösteren bir arayüzdür.

Ne işe yarar?

- Logları görsel olarak aramak ve incelemek,
- Dashboard'lar oluşturmak,
- Grafikler, tablolar, filtreler eklemek.
- Zaman bazlı log görüntüleme,
- Gerçek zamanlı log takibi,
- Özel dashboard'lar ile servis izleme.

Teşekkürler!

Teknoloji dolu günler geçirmeniz dileği ile hoşçakalın !