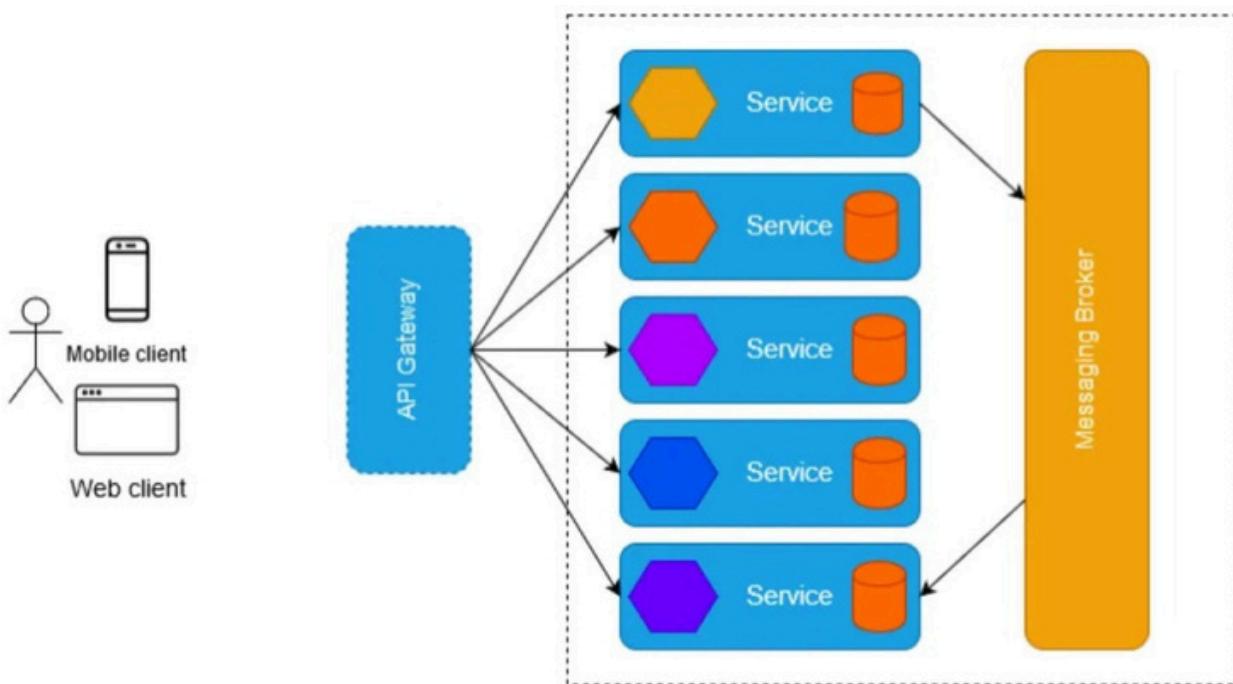


# MİKROSERVİS MİMARİSİ

FATİH SERHAT TURAN

# Mikroservis Mimarisi Nedir :

Mikroservis mimarisi, bir uygulamanın bağımsız, ölçeklendirilebilir ve tek bir işe odaklanmış atomik hizmetlere/servislere ayrılarak geliştirilmesini ve dağıtılmasını sağlayan bir yazılım geliştirme yaklaşımıdır. Bu yaklaşımında her bir servis, uygulamanın belirli bir işlevin sorumluluğunu üstlenir ve ihtiyaç doğrultusunda diğer servislerle iletişim kurarak daha büyük bir sistem oluşturur.



## Dağılımlılık Nedir :

Dağıtık olma, bir yazılımın farklı bileşenlerinin veya modüllerinin çeşitli bilgisayarlar veya sunucular üzerinde çalışacak şekilde tasarlanmasıdır. Mikroservis mimarisinde, uygulamanın tüm servislerini tek bir sunucuda çalıştırmak yerine, her bir servisin ayrı sunucularda bağımsız olarak çalışabilmesi bu dağıtık yapının temel özelliğidir. Dağıtık sistemlerde uygulamanın her bir servisi farklı bir konumda barındırılıyor olsa da, işlevsel açıdan bu servislerin bir bütün olarak çalışması esastır. Yazılımın mikroservis olarak tasarılanmasının yanında dağıtık bir özellikle olması; performans, yüksek ölçeklenebilirlik, daha iyi verimlilik ve yüksek kullanılabilirlik gibi avantajlar sunmaktadır.

## Ölçeklenebilirlik Nedir :

Yazılımın ölçeklenebilirliği, artan iş yükü veya kullanıcı sayısına rağmen sistemin performansını koruyarak sorunsuz çalışabilme yeteneğidir. Bu özellik, yazılım mimarisinin esnek ve modüler bir yapıda tasarılmamasını gerektirir. Böylece ihtiyaç duyulduğunda yatay (sunucu sayısını artırarak) veya dikey (donanım gücünü artırarak) büyümeye sağlanabilir. Ölçeklenebilir bir yazılım, değişen taleplere hızlıca uyum sağlar, işletme maliyetlerini optimize eder ve kullanıcı deneyimini kesintisiz sürdürür. Ölçekleme iki şekilde yapılabilir:

- Dikey ölçeklemede mevcut sunucuların işlemci, bellek veya depolama gibi donanım kaynakları artırılır.
- Yatay ölçeklemede ise birden fazla sunucu veya hizmet kopyası eklenerek yük dağıtilır.

Yatay ölçekleme genellikle mikroservis mimarisi, yük dengeleme (load balancing) ve bulut tabanlı altyapılarla desteklenir. Bu sayede sistem, performans düşüşü yaşamadan daha fazla kullanıcıya hizmet verebilir ve yüksek erişilebilirlik sağlanır.

### Uygulama ölçeklendirme stratejisini seçin



## Mikroservislerin Temel Prensipleri

### 1. Bağımsız Geliştirme ve Dağıtım

- Her mikroservis, kendiişlevselliliğinin ve veritabanını yöneten bağımsız bir birim olarak tasarılanır.
- Servislerin bağımsızlığı, farklı ekipler tarafından ayrı ayrı geliştirilebilmesine ve dağıtılabilmesine olanak tanır.

### 2. Farklı Teknolojilere Uygunluk

- Mikroservisler birbirinden bağımsız olduğu için, her servis farklı programlama dilleri ve teknolojiler kullanılarak geliştirilebilir.
- Bu esneklik, her servis için en uygun teknolojinin seçilmesine ve daha verimli çözümler üretilmesine imkan verir.

### 3. İşlevsellik Bazlı Yapılandırma

- Uygulama, işlevsel açıdan anlamlı parçalara bölünerek servisleştirilir. Her servis, belirli bir iş sürecine odaklanarak daha kolay yönetilebilir ve sürdürülebilir bir yapı sunar.

### 4. Gevşek Bağlantılı (Loose Coupling) İletişim

- Mikroservisler genellikle birbirleriyle **asenkron iletişim** kurar. Bu iletişim: Mesaj kuyruğu (message broker) sistemleri üzerinden sağlanabilir. HTTP tabanlı REST API çağrılarıyla da gerçekleştirilebilir. Bu yapı sayesinde servisler arasında sıkı bağımlılık oluşmaz ve ekipler servisler üzerinde bağımsız çalışabilir.

### 5. Bağımsız Ölçeklenebilirlik

- Her servis, ihtiyacı ve yük durumuna göre diğerlerinden bağımsız olarak yatay ya da dikey şekilde ölçeklenebilir. Bu yaklaşım, kaynakların verimli kullanılmasını ve performansın optimize edilmesini sağlar.

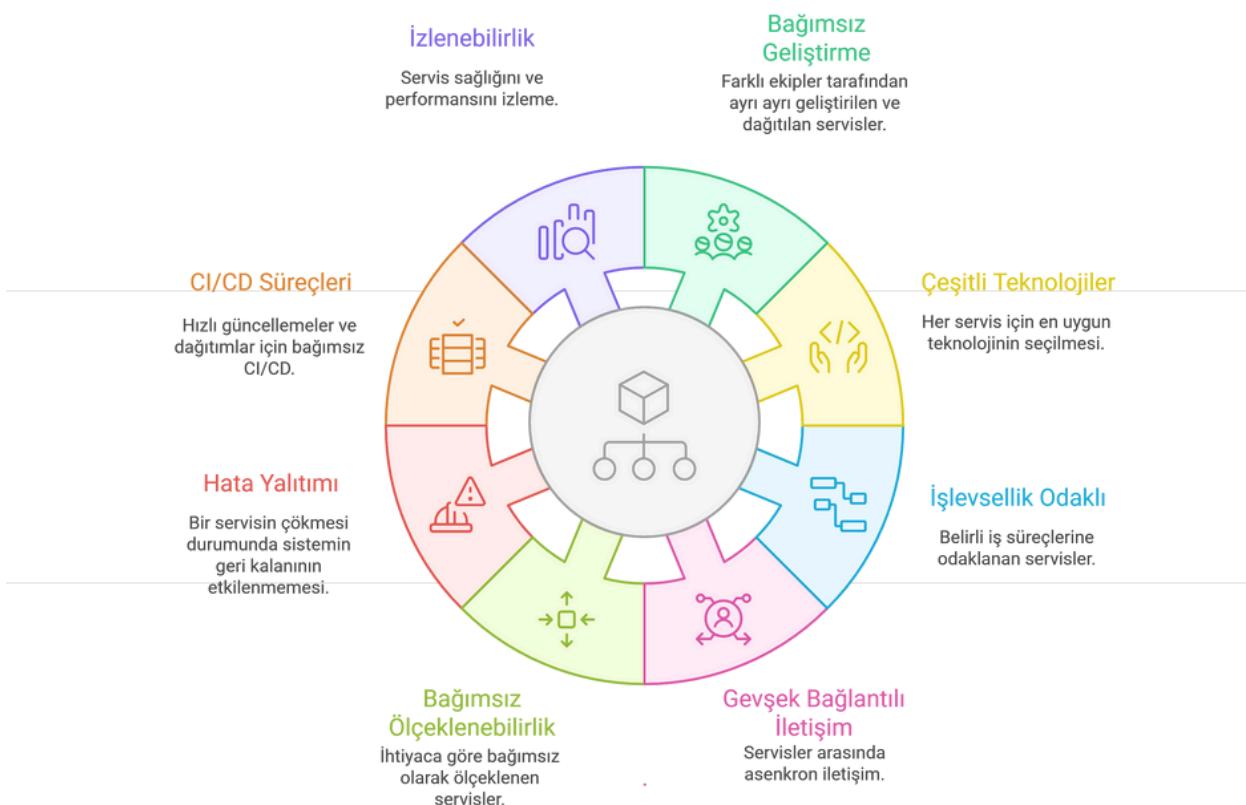
### 6. Hata Yalıtımı ve Dayanıklılık

- Birservisin çökmesidurumunda, sistemin tamamı bu durumdan etkilenmez. Servislerin izole yapısı, sistemin hata toleransını artırır ve genel bütünlüğü korur.

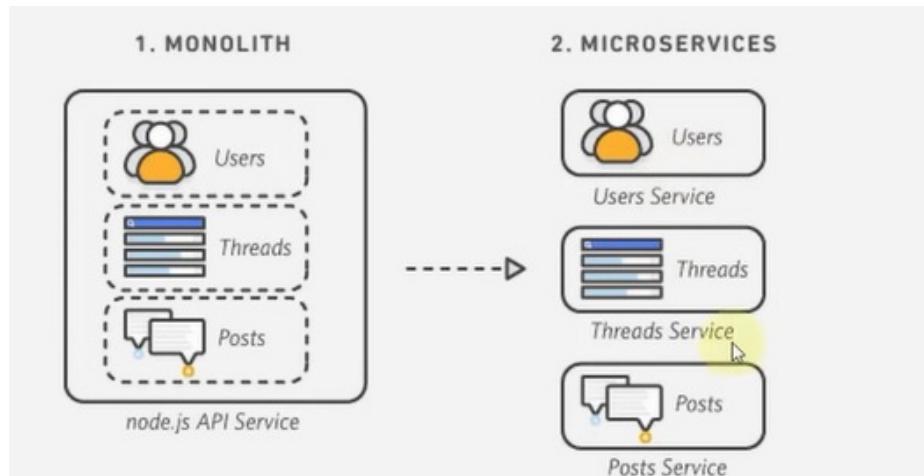
### 7. Sürekli Entegrasyon ve Sürekli Dağıtım (CI/CD)

- Mikroservisler, kendi bağımsız CI/CD süreçlerine sahip olabilir. Bu özellik, servislerin daha hızlı güncellenmesini, test edilmesini ve dağıtılmmasını sağlar.

## Mikroservis Prensipleri



## Monolitik VS Mikroservis :



## Monolitik vs Mikroservis :

Yazılım mimarisi tercihi, bir uygulamanın ölçeklenebilirliği, yönetilebilirliği, elistirme süreci ve operasyonel ihtiyaçları üzerinde doğrudan etkilidir. Bu bağlamda monolitik mimari ile mikroservis mimarisi arasında önemli farklılıklar bulunmaktadır. Aşağıda her iki mimarinin temel özellikleri, avantajları ve dezavantajları karşılaştırmalı olarak sunulmuştur:

### Avantajlar :

Konu	Monolitik Mimarî	Mikroservis Mimarîsi
<b>Basitlik ve Yönetim</b>	Tek kod tabanı olduğundan yapı ve dağıtım süreçleri daha basittir.	Her servis küçük ve odaklıdır; modüler yapı daha iyi ölçeklenir.
<b>İletişim Hızı</b>	Uygulama içi modüller arası iletişim doğrudan ve hızlidir.	Gevşek bağlılık sayesinde sistem geneline etki etmeyen hatalar mümkündür. Servisler bağımsız veri kaynaklarına sahip olabilir.
<b>Veri Tutarlılığı</b>	Tek veritabanı ile çalışıldığından veri yönetimi daha kolaydır. Düşüktür, bir hata tüm sistemi etkileyebilir.	Yüksek, bir servisteki hata genellikle diğerlerini etkilemez. Her servis bağımsız olarak ölçeklendirilebilir.
<b>Kusur Toleransı</b>	Uygulama bir bütün olarak ölçeklenir.	Her servis kendi CI/CD süreci ile bağımsız dağıtılabılır.
<b>Ölçeklenebilirlik</b>		
<b>Dağıtım Esnekliği</b>	Tüm sistem birlikte dağıtılır.	

## Dezavantajlar :

Konu	Monolitik Mimarî	Mikroservis Mimarîsi Her servis kendi ihtiyaçlarına özel teknolojiyle geliştirilebilir.
<b>Esneklik Eksikliği</b>	Teknolojik çeşitliliğe izin vermez.	
<b>Yapı Güncellemeye Zorluğu</b>	Küçük bir değişiklik bile tüm sistemin yeniden dağıtılmmasını gerektirir.	Sadece ilgili servis güncellenir, sistemin geri kalanı etkilenmez.
<b>Kusur Toleransı Düşüklüğü</b>	Bir modüldeki hata tüm sistemini devre dışı bırakabilir.	Servisler birbirinden izole olduğundan sistem bütünlüğü korunur.  Dağıtık yapı, izleme, loglama, hata yönetimi gibi süreçleri zorlaştırır.
<b>Operasyonel Karmaşıklık</b>	Operasyonel süreçler daha basit olabilir.	Servisler arası iletişim (HTTP, mesaj kuyrukları) ek gecikme yaratabilir.
<b>Servis İletişimi</b>	Dahili metod çağrılarıyla hızlı iletişim sağlanır.	

## Sonuç :

**Monolitik mimari**, küçük ve orta ölçekli projelerde daha hızlı başlangıç, düşük operasyonel karmaşıklık ve sade dağıtım avantajları sağlar.

**Mikroservis mimarisi** ise büyük, ölçeklenebilir ve sürekli gelişen sistemler için esneklik, yüksek hata toleransı, bağımsız geliştirme ve dağıtım imkânları sunar.

Ancak mikroservis mimarisi; servis sayısının artması, servisler arası iletişim karmaşıklığı, merkezi olmayan veri yönetimi ve gözlemlenebilirlik gibi konularda daha fazla planlama ve altyapı gerektirir. **Doğru mimari tercihi, projenin ölçüği, ekip yapısı, teknik gereksinimler ve uzun vadeli büyümeye stratejileri dikkate alınarak yapılmalıdır.**

---

## Mikroservis Mimarisinin Tercih Edilmesi Gereken ve Edilmemesi Gereken Senaryolar :

### Mikroservis Mimarisinin Uygun Olduğu Durumlar :

#### Büyük ve Karmaşık Uygulamalar

- Geniş kapsamlı, çok modüllü, yüksek trafik alan sistemlerde mikroservisler daha esnek ve yönetilebilir bir yapı sunar.
- Örneğin; e-ticaret sistemlerinde ödeme, ürün, kullanıcı gibi modüllerin ayrı servisler olarak yönetilmesi sistem performansını artırır.

#### Teknoloji Çeşitliliği Gerektiren Durumlar

- Farklı operasyonların farklı programlama dilleri ve teknolojilerle geliştirildiği durumlarda mikroservisler teknoloji bağımsızlığı sağlar.

## Mikroservis Mimarisinin Uygun Olmadığı Durumlar :

### Küçük ve Basit Uygulamalar

- Mikroservis yapısı, küçük projeler için gereksiz karmaşıklık ve operasyonel yük doğurur. Bu gibi durumlarda monistik mimari daha uyundur.

### Teknoloji ve Altyapı Kaynaklarının Yetersiz Olduğu Durumlar

- Mikroservisler gelişmiş bir altyapı ve DevOps kültürü gerektirir. Bu imkanlar yoksa sistemin sürdürülebilirliği zorlaşabilir.

### Hızlı Geliştirme ve Prototipleme Süreçleri

- Kısa sürede MVP (minimum viable product) ya da prototip geliştirmek gerekiyorsa mikroservis mimarisi zaman kaybına neden olabilir.

### Modüller Arası Sıkı Bağlılık Gerektiren Sistemler

- Uygulama bileşenleri arasında yoğun veri paylaşımı veya senkron işlemler varsa, modüllerin ayrıştırılması yerine monistik yapı tercih edilmelidir.

# Temel Kavramlar | Organizasyon Modelleri | API Gateway :

Mikroservis mimarisi, yazılım sistemlerini küçük, bağımsız ve birbirile uyumlu servisler hâlinde yapılandırmayı esas alır. Bu mimariyi oluşturan başlıca temel kavramlar aşağıda açıklanmıştır:

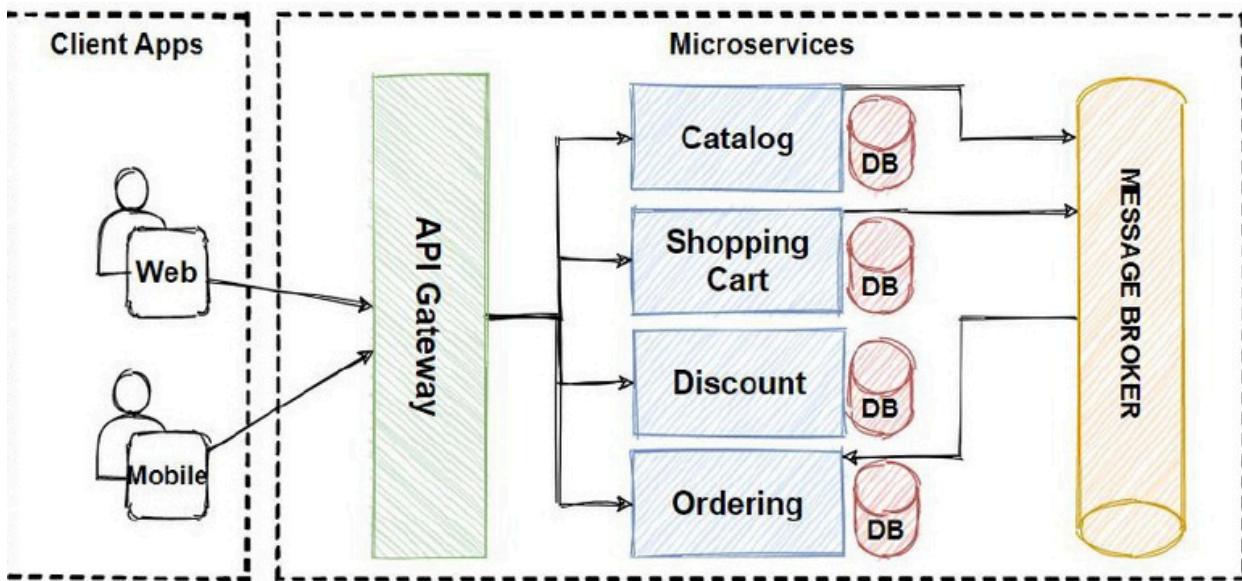
**Servis:** Mikroservis mimarisinin en temel yapı taşıdır. Belirli bir işlevi yerine getiren, bağımsız olarak çalışan, kendi veritabanına, iş mantığına ve uç noktalarına (endpoint) sahip modüler bir uygulama bileşenidir. Her servis, yalnızca tek bir iş alanına odaklanacak şekilde tasarılanır.

**İşlev (Functionality):** Her servisin gerçekleştirdiği özel operasyon ya da görevdir. Örneğin, kullanıcı yönetiminden sorumlu bir servis, kullanıcı yetkilendirme işlemlerini gerçekleştirmeye sahip olabilir. Bu işlevler, servisin sunduğu temel hizmetleri tanımlar.

**Bağımsızlık:** Mikroservislerin en önemli özelliklerinden biri olan bağımsızlık, her servisin kendi başına çalışabilir ve yönetilebilir olması anlamına gelir. Bir servisin arızalanması, güncellenmesi veya yeniden dağıtılması diğer servisleri etkilemez. Bu yapı sayesinde geliştirme, dağıtım ve bakım süreçleri daha esnek, yönetilebilir ve ölçülebilir hâle gelir.

**API (Application Programming Interface):** Servislerin birbirleriyle veya dış sistemlerle iletişim kurmasını sağlayan arayüzlerdir. Genellikle senkron iletişim modeline dayanır ve RESTful, GraphQL gibi protokoller aracılığıyla gerçekleştirilir.

**Mesaj Kuyruğu (Message Broker):** Servisler arası asenkron iletişimini sağlayan mesajlaşma altyapısıdır. Message broker sistemleri, servislerin doğrudan birbirine bağlı olmadan veri alışverişinde bulunmasına olanak tanır. Bu yapı sayesinde servisler, zamanlama ve performans açısından daha esnek bir ekilde çalışabilir.



## Mikroservis Organizasyon Modelleri :

Mikroservis mimarisi, organizasyonların yapısal özelliklerine ve uygulama gereksinimlerine göre farklı yaklaşımlar benimseyen, esnek bir yazılım geliştirme metodolojisidir. Bu mimari yaklaşımın, projelerin karmaşaklılığı ve organizasyonların ihtiyaçları doğrultusunda çeşitli stratejiler uygulanabilmekte olup, genellikle Teknoloji Odaklı Model, İş Odaklı Model, Veri Odaklı Model ve Karışık Model olmak üzere dört temel model türü üzerinden sistemler tasarılanmaktadır. Bu modeller, organizasyonların teknolojik altyapısı, iş süreçleri ve veri yönetimi stratejilerine uygun olarak seçilmekte ve uygulanmaktadır.

## Teknoloji Odaklı Model :

Teknoloji Odaklı Model, mikroservis mimarisinde her bir servisin belirli bir teknoloji yiğini ve dili etrafında geliştirildiği yapısal bir yaklaşımındır. Bu modelde, servislerin tasarımı ve geliştirilmesi sürecinde Java, Python, Go gibi farklı programlama dilleri ve teknolojik çözümler kullanılabilmekte, her servis kendi teknolojik özelliklerine göre optimize edilmektedir. Geliştirici ekiplerin teknoloji seçiminde özgürlük sahibi olması, servislerin performans ve verimlilik açısından en uygun teknolojik altyapı ile desteklenmesini sağlamaktadır.

Bu yaklaşımın temel avantajı, ekiplerin sahip oldukları teknolojik uzmanlık alanlarından maksimum düzeyde yararlanabilmeleri ve farklı teknolojilerin bir arada kullanılması ile ortaya çıkan sinerjilerden faydalananabilmeleridir. Ancak, çok sayıda farklı teknolojinin bir arada kullanılması, sistem yönetimi ve bakım süreçlerinde karmaşaklık yaratıbmaktadır, bu nedenle teknoloji çeşitliliğinin kontrollü bir şekilde yönetilmesi kritik önem taşımaktadır. Node.js, Java, Python gibi farklı teknolojilerin mevcut olduğu bir ortamda, ekiplerin bu çeşitliliği etkin bir şekilde koordine etmesi sistem bütünlüğü açısından hayatı öneme sahiptir.

## Veri Odaklı Model :

Veri Odaklı Model, mikroservis mimarisinde her bir servisin belirli bir veri yapısı veya veri grubu üzerinde yoğunlaştiği ve bu veri kaynaklarının yönetiminden sorumlu olduğu specialization yaklaşımıdır. Bu modelde, müşteri verileri yöneten bir servis, ürün kataloğunu yöneten bir servis gibi farklı veri domenlerine odaklanan yapılar oluşturulmakta ve her servis kendi veri alanının tüm işlemlerinden sorumlu olmaktadır. Servislerin veri bütünlüğü ve veri işleme performansını artırması, sistem genelinde daha etkin veri yönetimi sağlamaktadır.

Bu yaklaşımın temel avantajı, veri işleme süreçlerinin optimize edilmesi ve her servisin kendi veri yapısına uygun teknolojik çözümler kullanabilmesidir. Ancak, veri odaklı modelin uygulanmasında kritik zorluklar da bulunmaktadır; özellikle servislerin tüm veri yapılarını ilgilendiren işlemler söz konusu olduğunda, servisler arası veri senkronizasyonu ve tutarlılığın sağlanması karmaşık hale gelebilmektedir. Bu nedenle, veri bütünlüğünün korunması, servisler arası bağımlılığın artırılabilmesi ve veri güncelleme süreçlerinin koordinasyonu gibi faktörler, bu modelin başarılı implementasyonu için dikkatli bir planlama ve yönetim gerektirmektedir.

## Karışık Model :

Karışık Model, mikroservis mimarisinde farklı yaklaşımların bir kombinasyonunu kullanan hibrit bir stratejidir. Bu modelde, organizasyonlar iş odaklı ana yapı üzerine kurulu sistemlerini farklı teknolojilere sahip alt servisleri ekleyerek projenin tasarımını genişletebilmekte ve çeşitli ihtiyaçlara uygun esnek çözümler geliştirebilmektedir. Model, hem teknoloji odaklı hem de iş odaklı yaklaşımının avantajlarını bir araya getirerek, organizasyonların spesifik gereksinimlerine göre özelleştirilebilir bir yapı sunmaktadır. Unutulmamalıdır ki her organizasyonun ihtiyaçları ve uygulama alanları farklı olacağı için en uygun modelin belirlenmesi gereklidir. Aksi takdirde mimarinin tasarım başlangıcı olası hatalar ve öngörülemezliklerle ilerleyebilir.

## Mikroservislerde Servisler Arası İletişim Nasıl Gerçekleşir :

Mikroservis mimarisi doğası gereği farklı servislerin birbirileyle iletişim kurmasını gerektirir. Servisler arası iletişim genellikle aşağıdaki yöntemlerden biri veya birkaçının kombinasyonuyla gerçekleşir.

### HTTP Tabanlı API'lar :

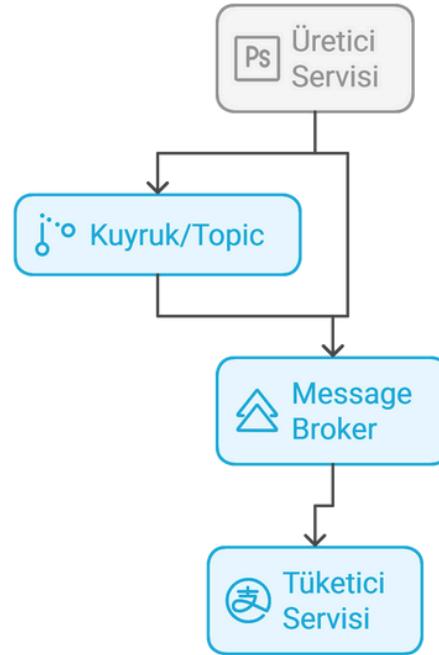


HTTP protokolü tabanlı REST API iletişimini, mikroservis mimarilerinde en yaygın kullanılan haberleşme yöntemlerinden biridir. Bu model, servislerin standart HTTP metodları (GET, POST, PUT, DELETE) aracılığıyla birbirileyle etkileşim kurmasını sağlar. İletişim süreci, istemci servisinin hedef servise HTTP talebi göndermesiyle başlar. Hedef servis, gelen talebi işledikten sonra uygun HTTP yanıt kodu ve veri ile karşılık verir. Bu yaklaşım, istemcinin servisin yanıtını beklemesi nedeniyle senkron iletişim modeline dayanır ve gerçek zamanlı veri alışverişi gerektiren senaryolarda tercih edilir.

Servisler arası veri transferi genellikle JSON formatında gerçekleştirilir, ancak XML gibi diğer yapılandırılmış veri formatları da desteklenir. REST mimarisinin stateless (durumsuz) yapısı ve HTTP protokolünün yaygın kabul görmüş standartları, bu yöntemin endüstri genelinde benimsenmesinin temel nedenleridir. Bu iletişim modelinin avantajları arasında kolay test edilebilirlik, platform bağımsızlığı ve mevcut web altyapısı ile uyumluluk yer alırken, senkron yapısından kaynaklanan performans darboğazları ve servis bağımlılıkları dezavantajları olarak değerlendirilebilir.

## Message Broker :

### Message Broker Sistemi



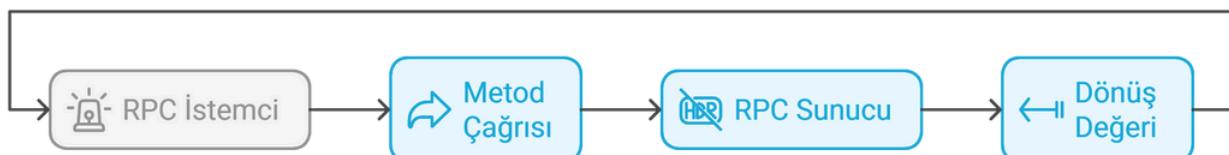
Mesaj aracı (Message Broker) tabanlı iletişim, mikroservis mimarilerinde asenkron haberleşme sağlayan önemli bir yöntemdir. Bu model, servislerin birbirlerine doğrudan bağımlı olmadan mesaj alışverişi yapmasına olanak tanır.

İletişim süreci, üretici servisin mesajı message broker'a göndemesiyle başlar. Mesaj, kuyruk (queue) veya konu (topic) yapıları içerisinde saklanır. Tüketici servis, mesajı uygun zamanda broker'dan alarak işler. Bu yaklaşım, servislerin farklı hızlarda çalışabilmesini ve geçici kesintilere karşı dayanıklı olmasını sağlar. Popüler message broker teknolojileri arasında Apache Kafka, RabbitMQ ve Amazon SQS yer almaktadır. Veri formatı genellikle JSON, XML veya binary yapılarında olabilir.

Bu modelin temel avantajları arasında yüksek ölçeklenebilirlik, servis bağımsızlığı ve hata toleransı bulunur. Ancak mesaj sıralaması ve eventual consistency konularında dikkatli tasarım gerektirir. Özellikle yüksek throughput gerektiren ve gerçek zamanlı yanıt beklenmediği senaryolarda tercih edilen bir yaklaşımındır.

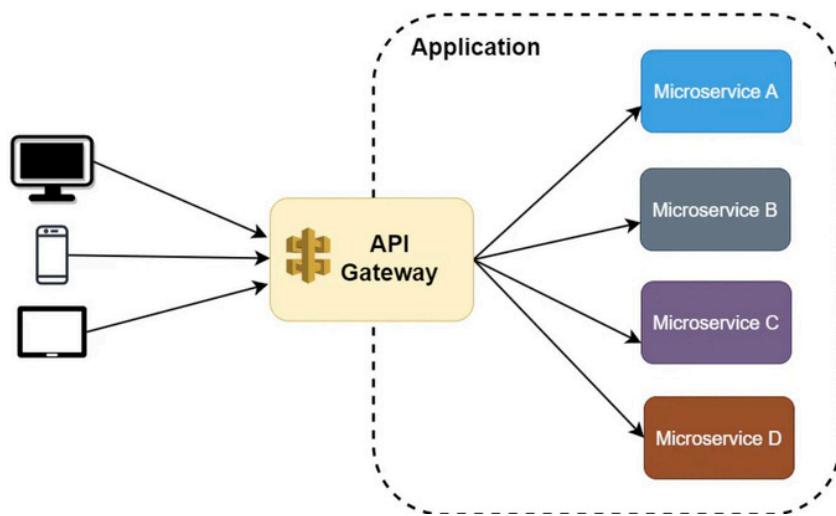
## RPC(Remote Procedure Call) :

### RPC İletişim Akışı



RPC (Remote Procedure Call), servislerin uzak konumlardaki prosedürleri yerel fonksiyon çağrıları gibi kullanmasını sağlayan senkron bir iletişim protokolüdür. Bu model, istemci servisin uzak sunucudaki metodları doğrudan çağrıarak sonuçları gerçek zamanlı olarak almasına olanak tanır. Bu yaklaşımın temel avantajları arasında düşük latency, tip güvenliği ve kolay kullanım yer almaktır. Ancak, servislerin sıkı bağlanması (tight coupling) nedeniyle sistem esnekliği azalabilir ve network kesintilerinde hata yönetimi karmaşıklaşır.

## Servisler Arası İletişimde API Gateway'ın önemi :



Modern yazılım geliştirme süreçlerinde mikroservis mimarileri, uygulamaların ölçeklenebilir ve esnek bir yapıda tasarılanmasını sağlayan önemli bir yaklaşım haline gelmiştir. Ancak, birçok küçük servisten oluşan bu dağıtık sistemlerde, client uygulamalarının farklı servislerle iletişim kurması, kimlik doğrulama süreçlerinin yönetilmesi ve trafik kontrolü gibi operasyonel zorluklar ortaya çıkmaktadır. Bu noktada API Gateway, mikroservis ekosisteminin karmaşıklığını yöneten ve sistem bütünlüğünü koruyan kritik bir bileşen olarak devreye girer. API Gateway, tüm mikroservislerin önünde konumlanarak, client isteklerini uygun servislere yönlendiren, güvenlik protokollerini uygulayan ve sistem performansını optimize eden merkezi bir kontrol noktası işlevi görür.

**API Gateway'in Temel İşlevi:** API Gateway, mikroservis mimarilerinde geliştirme sürecini hızlandıran ve uygulamaları merkezi bir nokta üzerinden haberleşirmeyi sağlayan, kullanıcılar için odaklanacağı muhim bir rol üstlenen bileşendir.

**Merkezi Yönetim Avantajları:** Böylece tüm servisler için tek bir API sunarak, client'ların birden fazla servise bağlanmak yerine merkezi bir API yöneticisinin sağladığı isteklerini gerçekleştirmesini sağlamaktadır.

**Kimlik Doğrulama ve Yetkilendirme:** Ayrıca sisteme kurulan merkezi API yapılanması sayesinde gerekli kimlik doğrulama ve yetkilendirme sorumluluklarında da üststerek servislerin de güvenliğini sağlayabilmektedir.

**Client Yönetimi:** API Gateway, client'ın yapacağı istek süreçlerinde hedef servisin hangi servise konumunu/ipsini/adresini öğrenmesine gerek kalmaksızın, tek bir merkezi nokta üzerinden istedeklerin doğru servise yönlendirilmesini sağlamaktadır.

**Trafik Yönetimi ve Optimizasyon:** Tüm isteklerin dışında, ölçeklendiriliş servisler üzerinde gelen trafiği dengeli bir şekilde yönlendirerek yük dağılıminin (load balancing) optimize edilebilir. Bu sayede, gelen tüm trafiği merkezi bir şekilde denetleyerek uygulama bütünü için gerekli güvenlik protokollerinin uygulanması merkezleştirilebilir.

# **Tasarım İlkeleri ve Veritabanı Stratejileri :**

## **Mikroservis Mimarisi Tasarımı Nasıl Gerçekleştirilir?**

Mikroservis mimarisi tasarımlı, projenin gereksinimleri, ölçügeye karmaşıklığına bağlı olarak detaylı ve dikkat gerektiren bir süreçtir. Ancak genel hatlarıyla bir mikroservis mimarisi aşağıdaki temel adımlar izlenerek tasarlantırı:

### **Gereksinimlerin Anlaşılması :**

Tasarım sürecinin ilk adımı, projenin gereksinimlerinin net bir şekilde anlaşılmasıdır. Bu aşamada, sistemin fonksiyonel ve fonksiyonel olmayan gereksinimleri ile performans hedefleri tanımlanmalıdır. Ekip üyeleri, proje sahipleri ve kullanıcılarla etkin bir iletişim kurularak ihtiyaçların eksiksiz ve doğru bir şekilde anlaşıldığından emin olunmalıdır.

### **Mikroservislerin Belirlenmesi :**

Gereksinimlerin netleşmesinin ardından, uygulama işlevleri birbirinden ayırtılmalı ve bağımsız, özerk servisler halinde yapılandırılmalıdır. Her bir mikroservis, belirli bir işlevi yerine getirecek şekilde tanımlanmalı ve servis sınırları net biçimde çizilmelidir.

### **İletişim Protokollerinin Belirlenmesi :**

Servisler arası iletişim, mikroservis mimarisinin en kritik konularından biridir. Bu nedenle, mikroservisler arasındaki veri alışverişinin nasıl gerçekleşeceğini, yani kullanılacak iletişim protokollerini belirlenmelidir. İlgili senaryoya uygun olarak senkron ya da asenkron iletişim modelleri tercih edilmeli, sistemin ihtiyaçlarına en uygun çözüm sağlanmalıdır.

### **Veritabanı Stratejisi :**

Her mikroservisin kendi veritabanına sahip olması esastır. Bu doğrultuda, her bir servisin veritabanı stratejisi ayrı ayrı ele alınmalı ve kullanım senaryosuna en uygun model seçilerek tasarlantırılmalıdır. Böylece veri bağımsızlığı ve esneklik sağlanmış olur.

### **Güvenlik Tasarımı :**

Güvenlik, mikroservis mimarisinin temel yapı taşlarından biridir. Tüm sistem genelinde kimlik doğrulama (authentication), yetkilendirme (authorization), veri güvenliği ve ağ güvenliği gibi başlıklar kapsamlı bir şekilde ele alınmalıdır. Her servisin yetkisiz erişimlere karşı korunması için gerekli güvenlik önlemleri alınmalı ve uygun teknolojik altyapı oluşturulmalıdır.

### **Hata Yönetimi ve İzleme :**

Birbirinden bağımsız birçok servisin bir bütün oluşturduğu mikroservis mimarisinde, sistemin tamamı kesintisiz olarak izlenebilir olmalıdır. Olası hataların erken tespiti ve müdahalesi için merkezi loglama, izleme ve hata yönetimi sistemleri kurulmalıdır. Böylece, kullanıcı deneyimi kesintiye uğramadan hata yönetimi yapılabilir ve sistemin onarılabilirliği artırılır.

### **Ölçeklenebilirlik ve Yedekleme :**

Son kullanıcıya sunulacak sistemde beklenen yük artışları göz önünde bulundurularak ölçeklenebilir bir yapı tasarlanmalıdır. Gerektiğinde sistem kaynaklarının artırılabilmesi ve kesintisiz hizmet sağlanması için yedekleme ve felaket senaryoları detaylı şekilde kurgulanmalıdır.

## Test ve Sürüm Yönetimi :

Tasarım sürecinin sonunda, sistemin bütüne yönelik kapsamlı test senaryoları hazırlanmalıdır. Bunun yanı sıra, her bir mikroservisin bağımsız olarak test edilebileceği yapılar kurulmalı ve sürekli entegrasyon/sürekli teslimat (CI/CD) süreçleri ile sürüm yönetimi etkin bir şekilde sağlanmalıdır.

## Mikroservis Mimaride Veritabanı Stratejileri :

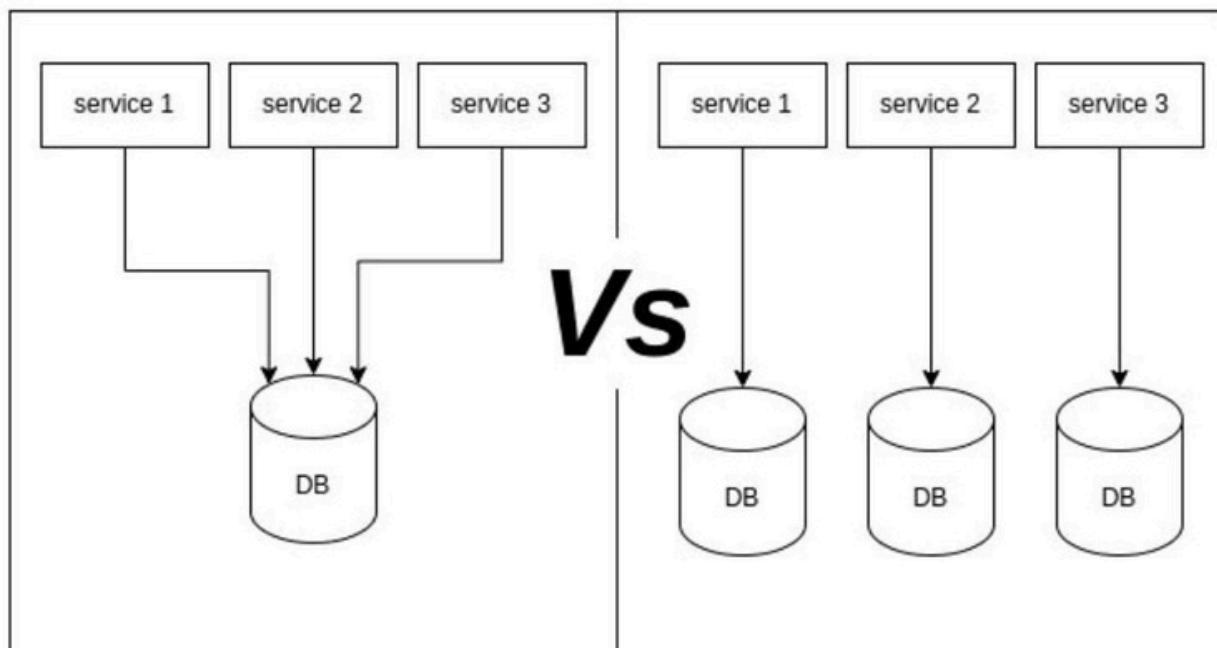
Mikroservis mimarisi birbiriyle bağımsız ve özerk servislerin bir araya getirilmesini içerdiginden , veritabanı stratejileri değişiklik gösterebilir. Her mikroservis kendi veritabanına sahip olabilir veya bazı durumlarda mikroservisler verileri paylaşmak için ortak bir veritabanı kullanabilirler.

### Her Mikroservis Kendi Veritabanına Sahip Olabilir :

Her mikroservis kendi özelveritabanına sahip olabilmektedir.Bu strateji mikroservicelerin bağımsızlığını en üst düzeye çıkarmaktadır ve bir mikroservisin veritabanındaki değişikliklerin diğerini etkilemesini önlemektedir. Bu stratejinin dezavantajları , bazı verilerin farklı veritabanlarında tekrarlanması veya istemsiz veri tutarsızlıklarının meydana gelmesi gibi sorunlar doğurabilmektedir.

### Ortak Veritabanı Kullanımı :

Tüm mikroseviciler , ortak bir veritabanı kullanarak da verileri paylaşabilirler . Bu strateji veri bütünlüğünü sağlamak ve veri tekrarı sorunlarını önlemek için avantajlı olabilir. Ancak bu strateji de servisler arasında sıkı bir bağlantıya neden olmaktadır ve diğer servislere direkt etkide bulunabilmektedir.



Database per service Vs Shared  
database

## **Veri Servisi :**

Mikroservisler, veri paylaşımı ve yönetimi için merkezi bir veri servisi kullanabilirler. Bu strateji, verilerin tek bir merkezden yönetilmesini sağlayarak her bir mikroservisin kendi veritabanını tutmasına gerek bırakmaz. Böylece veri tekrarının önüne geçilebilir. Ancak bu yaklaşım, sistemde ek bir karmaşıklık oluşturabilir ve performans açısından bazı zorluklara neden olabilir.

## **Event Sourcing :**

EventSourcing, verilerin doğrudan son hallerinin değil, bu verileri oluşturan olayların (event'lerin) kayıt altına alındığı bir veri modelidir. Mikroservis mimarisinde bu model, veri bütünlüğünün korunması ve veri tekrarlarının önlenmesi açısından etkin bir şekilde kullanılabilir. Her değişiklik bir olay olarak kaydedildiğinden, sistemin geçmiş durumu izlenebilir ve yeniden oluşturulabilir.

## **CQRS (Command Query Responsibility Segregation) :**

CQRS, veritabanı işlemlerini iki ayrı sorumluluk alanına ayıran bir tasarım desenidir: komutlar (command) veri üzerinde değişiklik yaparken, sorgular (query) yalnızca veri okuma işlemi gerçekleştirir. Mikroservisler bu yaklaşımı benimseyerek, veri yazma ve okuma süreçlerini birbirinden ayırrı. Bu ayırım, performans artışı ve daha iyi ölçeklenebilirlik gibi önemli avantajlar sağlar.

## **Servisler Arası İletişim Modelleri :**

Mikroservis mimarisinin, birbirinden bağımsız ve sadece tek bir işeodaklanmış tekil servislerin bir araya gelerek bir bütün olarak çalıştığı bir disiplindir, haliyle bu servislerin bir bütünü meydana getirebilmesi için kendi aralarında iletişim kurmaları gerekmektedir ve bu iletişim de senaryosuna göre doğru modelle inşa edilmelidir.

## **Servisler Arası Senkron ve Asenkron İletişim Modelleri :**

Mikroservis mimarisinde servisler arası iletişim modelleri teknik olarak senkron ve asenkron olarak değerlendirilir. Bu modeller, servisler arasındaki iletişim senaryosu ve iş kuralı gereği mantıken tercih edilerek uygulanmaktadır.

### **Mikroservis İletişim Modelleri**



Bir servis başka herhangi bir servisle iletişim kurmak istiyorsa bu işlem bir istek ve devamında da bu istekten dönen yanıt olarak gerçekleşmektedir. Burada iletişimın senkron veya asenkron olarak adlandırılmasının bu aşamada yürütülecek olan strateji ile ilgiliidir. Senkron iletişimde, isteği gönderen servis yanıt gelene kadar bekler; bu, genellikle HTTP tabanlı REST çağrılarında görülür. Bu yöntem basit ve doğrudan olsa da, yanıt süresine bağlı olarak servisler arasında gecikmelere neden olabilir. Asenkron iletişimde ise, isteği gönderen servis yanıt beklemeden işlemesine devam eder; bu, genellikle mesaj kuyrukları (örneğin RabbitMQ, Kafka) aracılığıyla gerçekleştirilir. Bu yaklaşım, servisler arasında daha gevşek bağlılık sağlar ve yüksek ölçeklenebilirlik sunar.

## Senkron İletişim :



**Senkron iletişim**, bir servisin diğer bir servisle doğrudan etkileşimde bulunduğu bir iletişim modelidir. İsteği gönderen servis, yanıt gelene kadar bekler ve ancak yanıt geldikten sonra işlemini tamamlar. Bu modele “senkron” denmesinin nedeni, hizmetler arasında bir senkronizasyon sürecinin bulunmasıdır. Bir servis diğer bir servisten yanıt almadığı sürece bir sonraki adıma geçemez ve beklemeye devam eder.

### Senkron İletişim: Avantajlar ve Dezavantajlar

<b>Avantajlar</b>	<b>Dezavantajlar</b>
<p>Senkron iletişim modeli basit yapısı sayesinde kolay anlaşılabilir. Bu durum, servisler arası iletişim sürecinin daha rahat takip edilmesine olanak tanır.</p> <p>Hata ayıklama ve izleme süreçleri doğrudan iletişim üzerinden yürütüldüğü için daha basit ve hızlıdır.</p>	<p>Servisler arası sıkı bağlılık, sistemin esnekliğini azaltır ve servislerden biri çalışmadığında tüm sistemin etkilenmesine neden olabilir.</p> <p>Bu sıkı bağlılık nedeniyle uygulamanın ölçeklenebilirliği sınırlanabilir ve yüksek trafikte performans sorunları ortaya çıkabilir.</p> <p>Yanıt süresi uzun olan servisler, kendisine bağlı tüm servislerin işlem akışını geciktirerek sistem performansını olumsuz etkileyebilir.</p>

## Asenkron İletişim :



Asenkron iletişimde bir servis diğer bir servisle iletişime geçerken bekleme olmadan mesaj gönderebilir ve işlemin tamamlanmasına gerek kalmadan diğer bir işleme geçebilir. Böylece işlemler paralel olarak gerçekleştirilebilir ve bir servis diğer bir servisin cevabını beklemeksızın çalışmasına devam edebilir.

## Asenkron İletişim: Avantajlar ve Dezavantajlar

<b>Avantajlar</b> Servisler birbirinden bağımsız çalışabildiği için sisteme yüksek esneklik ve düşük bağımlılık sağlanır. Yüksek trafik altında servisler, gelen talepleri kuyruklayarak sistemin çökmesini engelleyebilir ve daha yüksek performans sunabilir. Uygulama, servislerin cevap verme süresine bağlı kalmadan işlemlerine devam edebilir; böylece işlem akışında tıkanma yaşanmaz.	<b>Dezavantajlar</b> Karmaşık mesajlaşma yapıları nedeniyle hata ayıklama ve takip süreçleri daha zor ve zaman alıcı olabilir. Gerçek zamanlı yanıt beklenen senaryolarda, mesaj gecikmeleri kullanıcı deneyimini olumsuz etkileyebilir. Mesaj sıralaması, tekrar deneme (retry) mekanizmaları ve veri tutarlılığı gibi konular için ek altyapı ihtiyacı doğar.
--	---

## Senkron İletişim Örnekleri

### E-Ticaret Uygulaması Örneği

Bire-ticaret uygulamasında sepet ve stok işlemlerinin aynı mikroserviste olmaması durumunda varsayılmış. Eğer ki, müşteri sepete bir ürünü eklediğinde sepet servisi doğrudan stok servisini çağırarak ilgili ürüne dair stoğu güncellenmesini talep ediyor bu talep neticesini bekleyerek, ve stok servisi de bu talebe karşılık işlemi tamamlayarak cevap döndürüyor. Bu senkron bir iletişim senaryosudur.

### Banka Sistemi Örneği

Bankasistemlerinde müşterilerin kendi arasındaki transfer süreci senkron iletişim modeline örnek bir senaryodur. X müşteri tarafından Y müşterinin dair yapılmaya para transfer talebi, transfer işleminden sonraki mikroservise gönderilir ve gerekli hesap ve bakiye kontrollerü neticesinde transfer ya da gerçekleştirmeye ya da tutarsızlıktan dolayı iptal edilir. Her iki durumda da işlem sonucu response olarak kullanıcıya döndürülür.

## Asenkron İletişim Örnekleri

### E-Posta Sistemi Örneği

durumlarda, eğer ki, gönderilmek istenen e-posta isteği bir kuyruğa eklenip arasından da e-posta işlemleri sonuç alır. Kuyruktan mesaj alarak gönderme işlemi gerçekleştiriyor bilir asenkron iletişim senaryosudur. Böylece onlarca kullanıcıya bülten uygulamasının yaratması ve son kullanıcıları bekletmemek rahatça(asenkron bir şekilde) e-posta gönderilebilir.

### İstatistik Rapor Örneği

İletişim modeline örnek senaryo teşkil edebilmektedir. Kullanıcı rapor talebinde bulunduran zaman hal ile mesaj kuyruğuna gönderir ve rapor işlemlerinden sorumlu serviste bu kuyuktan mesajı alarak raporu işlemeye başlar ve tamamlandıgı zaman mail ya da arayüz üzerinden kullanıcıya bilgi verebilir.

### Dosya İşleme Örneği

Dosya işlemleri de genellikle asenkron iletişim modelinde yürütülür. Özellikle büyük dosyalar yüklenmek istediğiinde, mesaj kuyruğuna gerekli bilgilere sahip olan mesaj atılır, bu işten sorumlu olan serviste mesajı işleyerek dosyanın işlenmesini asenkron bir şekilde gerçekleştirir.

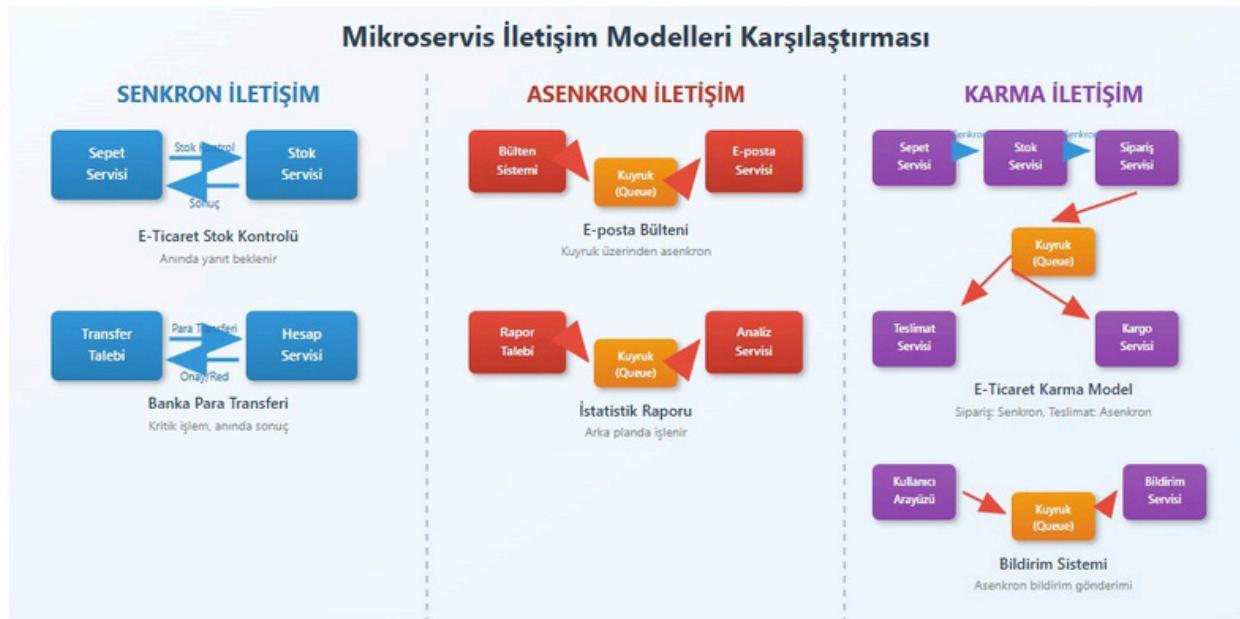
## Karma İletişim Örnekleri

### E-Ticaret Kompleks Sipariş Örneği

Bir e-ticaret uygulamasında sepet, sipariş, stok ve tesimat işlemlerinin ayrı mikroservislerde ele alındığını düşünelim. Müşteri bir sipariş verdiğiinde; sepet servisi, sipariş doğrulama servisi, stok envanteri sevsi bir şekilde çağrırlabilir ve stok güncellemesi sağlandıktan sonra sipariş tamamlanabilir. Ancak siparişin teslimat işlemi asenkron bir şekilde yürütülebilir ve bu sipariş dair bilgi bir mesaja broker'a atılarak teslimatla uğraşan servisler tarafından asenkron işlenerek sonucun kargo şirketinin yanıtına bağlı olarak işlenmesi sağlanabilir.

### Notification Sistemi Örneği

Uygulamalardaki notification sistemlerde asenkron iletişim modeli bir örnektir. Kullanıcı arayüz üzerinden bildirim talebinde bulunarak bu bildirim mesaj kuyruğuna ekler. Notification'dan sorumlu bir servis se kuyruktan mesajı alır ve mesajla birimlerin alıcıları gerekli bildirim çalışmasını gerçekleştirir.



## Mikroservislerin İzlenmesi ve Metriklerin Toplanması

Uygulama izleme, hangi mimari temel üzerine inşa edilmişolursa olsun, bir yazılım uygulamasının operasyonel durumunu, performans metriklerini ve potansiyel sorunlarını sürekli olarak takip eden kritik bir sistem yönetimi disiplinidir. Bu süreç, uygulama sağlığınıın değerlendirilmesi, performans anomalilerinin tespiti ve kullanıcı deneyiminde kesinti yaşanmaksızın proaktif müdahalelerin gerçekleştirilmesi için vazgeçilmez bir rol üstlenmektedir.

## **Uygulama İzlemenin Temel Faydaları**

### **Performans Anomalilerinin Erken Tespiti**

Uygulama izleme sistemleri, gerçek zamanlı performans metriklerini analiz ederek hizmet kesintilerine yol açabilecek performans degradasyonlarını önceden tespit etme kapasitesi sağlar. Bu proaktif yaklaşım, sistem güvenilirliğinin korunması ve kullanıcı memnuniyetinin sürdürülmesi açısından kritik öneme sahiptir.

### **Proaktif Sorun Yönetimi**

İzleme sistemleri, olası hataları ve performans düşüklüklerini öngörücü analitik yöntemlerle erkenden belirleme imkanı sunar. Bu proaktif yaklaşım, reaktif sorun çözme modelinden ziyade önleyici bakım stratejilerinin uygulanmasına olanak tanır.

### **Kullanıcı Deneyimi Optimizasyonu**

Modern kullanıcılar yüksek performans, güvenilirlik ve kesintisiz hizmet beklemektedir. Uygulama izleme, kullanıcı davranış analitiği ve sistem performans korelasyonu aracılığıyla kullanıcı deneyiminin sürekli optimizasyonuna katkı sağlar.

### **Hızlı Olay Müdahalesi ve Çözümlemesi**

Monitoring sistemleri, kritik olayların gerçek zamanlı tespiti ve otomatik eskalasyon mekanizmaları sayesinde ortalama çözüm süresini (MTTR ) Mean Time To Resolution) minimize eder. Bu durum, hizmet kesintilerinin etkisinin azaltılması ve operasyonel verimliliğin artırılması açısından hayatı önem taşır.

### **Güvenlik ve Anomali Tespiti**

İzleme sistemleri, güvenlik hallerinin ve sistem anomalilerinin tespitinde kritik rol oynar. Anormal trafik paternleri, yetkisiz erişimeleri ve potansiyel güvenlik açıklarını gerçek zamanlı olarak tespit edilebilir.

## **Mikroservislerde Monitoring'in Kritik Önemi**

Mikroservis mimarisi, dağıtık sistemlerin karmaşıklığı nedeniyle geleneksel monolitik uygulamalara kıyasla çok daha sofistik izleme stratejileri gerektirir. Dağıtık yapının her bir bileşeninin bağımsız olarak izlenmesi, servisler arası iletişim analiz edilmesi ve sistem genelindeki performans korelasyonlarının anlaşılması, mikroservis ekosisteminin optimal çalışması için vazgeçilmez gerekliliklerdir.

## **Loglama**

### **Yapilandırılmış Loglar (Structured Logging)**

Servislerin çalışma zamanlarındaki reflekslerini takip etmemizi ve yorumlamamızı sağlayan loglardır. Servisin sağlık durumunu ve işleyiş gidişatı sürecini anlayabilen hale getirmekte ve kolaylıkla analiz edebilmemizi sağlamaktadır.

## Merkezi Günlük Toplama (Centralized Log Collection)

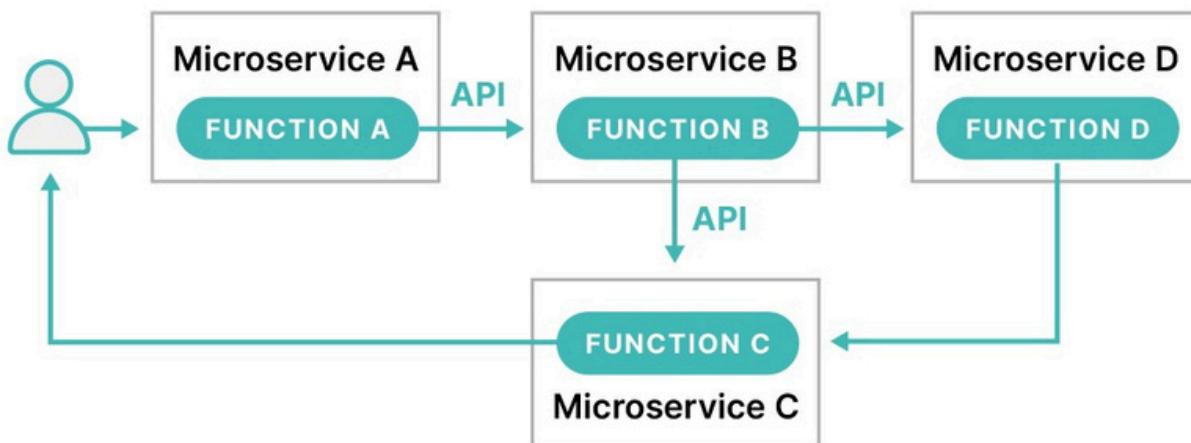
Tüm mikroservislerin ürettiği logları merkezi bir konumda toplamak ve genel değerlendirmeyi bu noktada gerçekleştirmektir. Elasticsearch gibi araçlar kullanılarak sağlanabilir.

## İzlenebilirlik (Traceability)

Client'tan gelen talebin işlenmesi sürecinde tüm servislerdeki akışın takip edilmesini ve böylece bir iş amacıyla gerçekleştirilen işlevin izini sürmesi sağlayan bir yöntemdir.

## Distributed Tracing :

Dağıtık izleme, mikroservis mimarisinin karmaşık yapısında kritik bir observability bileşeni olarak öne çıkmaktadır. Bu teknoloji, tek bir client talebinin mikroservis ekosistemi içerisinde geçirdiği tüm aşamaları, servisler arası iletişim süreçlerini ve her bir servisteki işlem adımlarını kapsamlı bir şekilde izleme ve analiz etme imkanı sağlar. Böylece, dağıtık sistemlerdeki veri akışının end-to-end görünürlüğü elde edilerek, sistem performansının bütünsel olarak değerlendirilmesi mümkün hale gelir. Büyük ölçekli ve karmaşık dağıtık sistemlerde, kullanıcı taleplerine yanıt verme sürecinde onlarca hatta yüzlerce mikroservis etkileşime girebilmektedir. Bu kompleks yapıda, geleneksel monitoring yaklaşımı yetersiz kalırken, dağıtık izleme teknolojisi ile her bir işlemin detaylı analizi gerçekleştirilebilir. Bu kapsamlı izleme yaklaşımı, sistemin atomik seviyede incelenmesine olanak tanıyar, performans optimizasyonu ve sorun giderme süreçlerinde kritik veriler sağlamaktadır.



Yukarıdaki görselden de anlaşılacağı üzere bu izleme, isteğin yola çıktığı noktadan başlayarak tüm yol boyunca hangi servislerde etkilendiğini, ne kadar süre harcadığını ve işlevsel olarak yapılan tüm manipülasyonları anlamamıza olanak tanımaktadır.

# Mikroservislerin Güvenliği :

Mikroservislerin güvenliği, dağıtık sistem mimarilerinde kritik önemesahip bir konudur. Yapısal olarak karmaşık ve dağıtık bir yapıya sahip olan mikroservis mimarisinin güvenliğini sağlamak için özelleşmiş güvenlik stratejilerine ihtiyaç duyulmaktadır. Mikroservislerin güvenliğini sağlamak için dikkate alınması gereken temel güvenlik bileşenleri şunlardır:

## Kimlik Doğrulama ve Yetkilendirme :

Mikroservislere erişim sağlayan kullanıcıların kimlik doğrulaması gerçekleştirilmeli ve kullanıcı bilgileri güvenli bir şekilde korunmalıdır. Erişim kontrolü prensibine uygun olarak, yalnızca gerekli yetkilere sahip kullanıcıların ilgili kaynaklara erişimi sağlanmalıdır. Bu kimlik doğrulama işlemleri için kullanılabilecek standart yöntemler arasında JWT (JSON Web Token) tabanlı doğrulamalar, OAuth 2.0 ve OpenID Connect gibi endüstri standartı protokoller bulunmaktadır..

## Veri Güvenliği :

Mikroservisler arasında iletilen verilerin bütünlüğü ve gizliliği sağlanmalıdır. Bu kapsamda:

**İletişim Şifreleme:** HTTPS gibi güvenli iletişim protokolleri kullanılarak veri akışı şifrelenmelidir.

**Ağ Güvenliği:** Mikroservisler arası iletişim ağının güvenliği, sanal ağlar (VPN), güvenlik duvarları (firewall) ve ağ segmentasyonu gibi yöntemlerle siber saldırılara karşı korunmalıdır

**Veri Maskeleme:** Hassas veriler için uygun maskeleme ve anonimleştirme tekniklerinin uygulanması önerilmektedir

## API Güvenliği :

Mikroservislerindən dünya ile etkileşim kurduğu ana noktalar API'lardır. API güvenliği için:

- JWT tabanlı doğrulama mekanizmaları implement edilmelidir
- Rate limiting ve throttling politikaları uygulanmalıdır
- API versiyonlama stratejileri güvenlik odaklı olarak tasarılanmalıdır

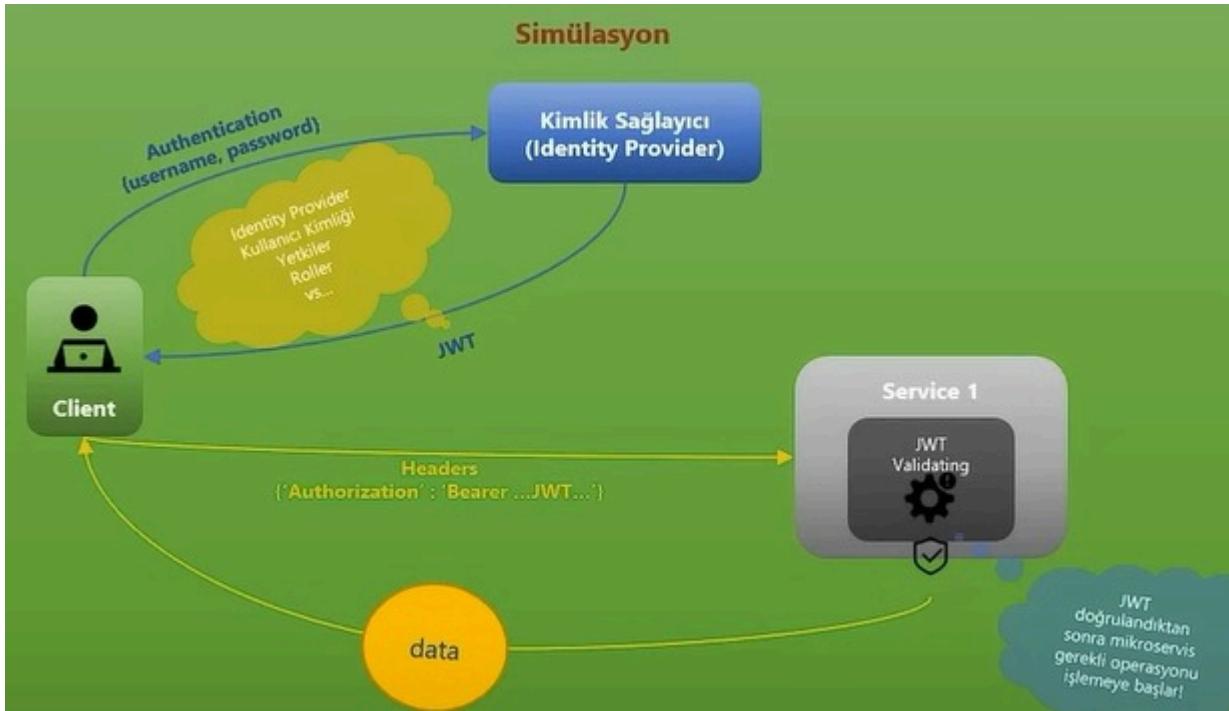
Mikroservislerde güvenliğin temel unsurlarından biri **izlenebilirlik** ve sistem davranışlarındaki anomalilikleri gerçek zamanlı olarak tespit edebilecek **logging ve monitoring** mekanizmalarıdır. Bu yapı sayesinde anormal aktiviteler proaktif olarak tespit edilebilir ve güvenlik ihlalleri minimize edilebilir.

## Mikroservislerde Kimlik Doğrulama ve Yetkilendirme :

Mikroservis mimarisinde kimlik doğrulama ve yetkilendirme, güvenlik mimarisinin temel taşılarından biridir. Bu süreçte kullanabileceğimiz başlıca yaklaşım şunlardır:

### Token Tabanlı Kimlik Doğrulama (JWT) :

JWT, mikroservislerarasındaki kullanıcı kimliğine yetkilendirme bilgilerini güvenli bir şekilde taşıyan ve doğrulamayı sağlayan popüler bir protokoldür.



JWT'nin en güçlü yönlerinden biri, servisler arası kimlik bilgilerinin paylaşılması gereken senaryolarda sağladığı esnekliktir. Bir kez kimlik doğrulama işlemi gerçekleştirildikten sonra, JWT token'ı mikroservisler arasında güvenli veri taşıyıcısı olarak kullanılır. Bu yaklaşım, dağıtık sistemlerde kimlik bilgilerinin tutarlı ve güvenli bir şekilde aktarılmasını sağlar.

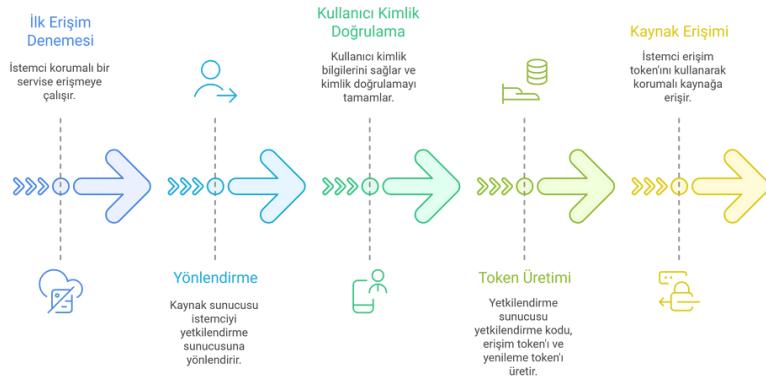
## OAuth 2.0 :

OAuth 2.0, uygulamaların kaynak sahiplerinin verilerine güvenli ve kontrollü bir şekilde erişim sağlamasını mümkün kıyan endüstri standartı bir yetkilendirme protokolüdür. Bu protokolün temel prensibi, tüm kimlik doğrulama ve yetkilendirme sorumluluklarının merkezi bir yetkilendirme sunucusu (Authorization Server) tarafından üstlenilmesidir.

### OAuth 2.0'in Temel Bileşenleri:

- Resource Owner:** Kaynak sahibi (genellikle son kullanıcı)
- Client:** Korunan kaynağa erişmeye çalışan uygulama
- Authorization Server:** Kimlik doğrulama ve yetkilendirme işlemlerini gerçekleştiren sunucu.
- Resource Server:** Korunan kaynakları barındıran sunucu

OAuth 2.0 Yetkilendirme Kodu Akışı



Erişim hedefi olan kaynak sunucusu (Resource Server) tarafından, Authorization Server'a gerekli yetkilendirme bilgileri aktarılır. Bu süreç, standartlaştırılmış bir protokol akışı içerisinde gerçekleşir.

#### **OAuth 2.0 Authorization Code Flow Örneği:**

Bir istemcinin (client) korumalı bir servise erişmek istediği senaryoyu ele alalım:

### **1. İlk Erişim Denemesi ve Yönlendirme**

Kaynak sunucusu, yetkisiz bir erişim girişimi tespit ettiğinde, gelen isteği Authorization Server'a yönlendirir. Bu yönlendirme işlemi HTTP 302 redirect response'u ile gerçekleştirilir. Kullanıcı, kimlik doğrulama için Authorization Server'ın login endpoint'ine yönlendirilir.

### **2. Kullanıcı Kimlik Doğrulama**

Authorization Server, kullanıcıdan kimlik bilgilerini (credentials) talep eder. Kullanıcı, geçerli kimlik bilgilerini (kullanıcı adı/şifre, multi-factor authentication vb.) sağlar. Kimlik doğrulama işlemi başarıyla tamamlanır.

### **3. Token Üretimi ve Dağıtımı**

Authentication sürecinin başarıyla tamamlanması durumunda, Authorization Server aşağıdaki token'ları üretir:

- **Authorization Code:** Kısa süreli, tek kullanımlık kod
- **Access Token:** Kaynak sunucusuna erişim için kullanılacak token
- **Refresh Token:** (opsiyonel): Access token'in yenilenmesi için kullanılan long-lived token

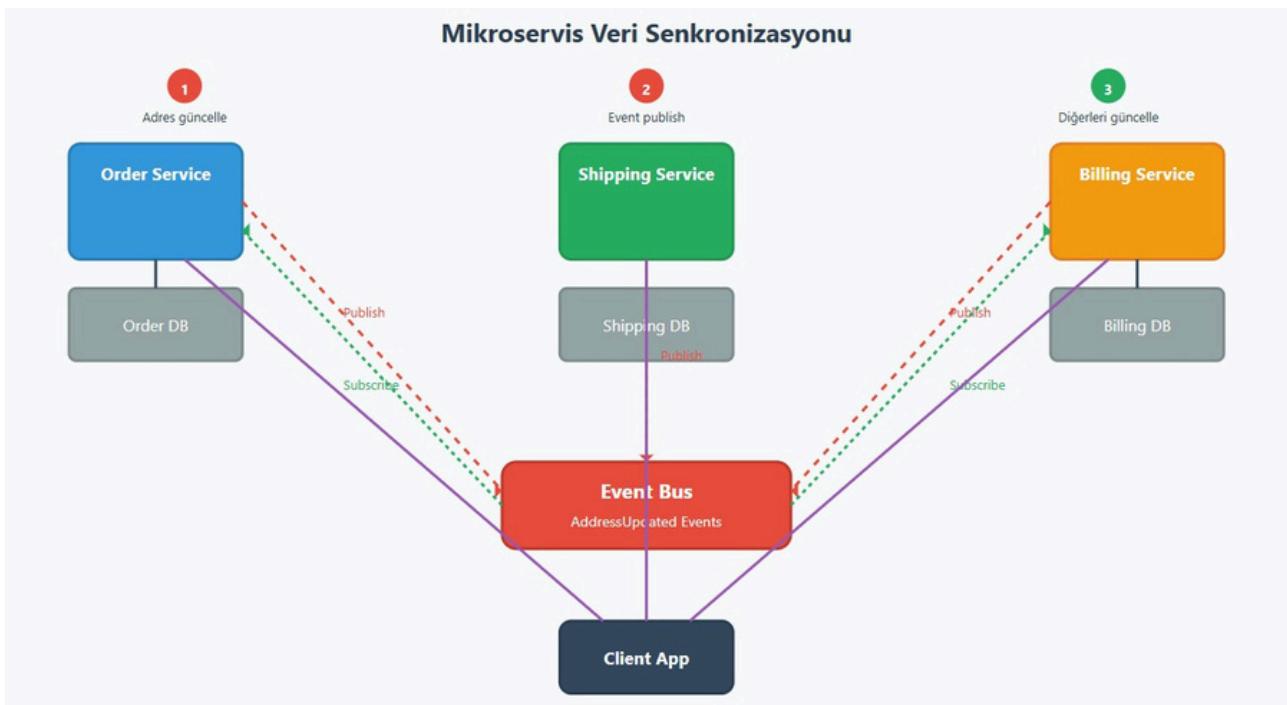
### **4. Kaynak Erişimi**

Access token, hedef servise erişim için uygun scope ve permission'lara sahipse, kullanıcı başarıyla kaynak sunucusuna erişim sağlayabilir. Her API çağrısında, access token Authorization header'ında Bearer token olarak ilettilir. Kaynak sunucusu, gelen token'i validate ederek erişim kararını verir.

## **Servisler Arası Veri Senkronizasyonu**

**Veri senkronizasyonu**, birden fazla sistem arasında verilerin güncel ve tutarlı kalmasını sağlayan kritik bir süreçtir. Mikroservis mimarisinde uygulama, birbirinden bağımsız servisler halinde yapılandırıldığından, bu servisler arasındaki veri senkronizasyonu büyük önem taşır. Mikroservis mimarisinde en iyi uygulama (best practice) olarak her servis kendi veritabanına sahip olmalıdır. Ancak bu durum, servisler arasında aynı verinin bütünsel olarak tutarlığını sağlamak zorunlu kılar.

Örneğin; üç farklı servisin ve her birinin kendine ait üç ayrı veritabanının bulunduğu bir senaryoyu ele alalım. Bu veritabanlarında bir müşterinin adres bilgisi saklanıyor olsun. Eğer bir servis, bir müşteriye ait adres bilgisini güncellerse, diğer iki veritabanında da bu bilginin eş zamanlı olarak güncellenmesi gereklidir. Benzer şekilde, bir müşterinin bilgileri **Müşteri Hizmetleri Servisi** üzerinden değiştirildiğinde, bu değişikliğin **Sipariş Servisi** gibi diğer servisler tarafından da anlık olarak görüntülenebilir olması gereklidir.



## API Çağrıları

Mikroservisler arasındaki veri senkronizasyonunu sağlamak amacıyla **API çağrıları** kullanılabilir. Bu yaklaşım, bir servis diğer bir servisin API'sini tetikleyerek gerekli veri güncellemelerini başlatır. Böylece, servisler arası doğrudan iletişim kurularak veri tutarlılığı korunur.

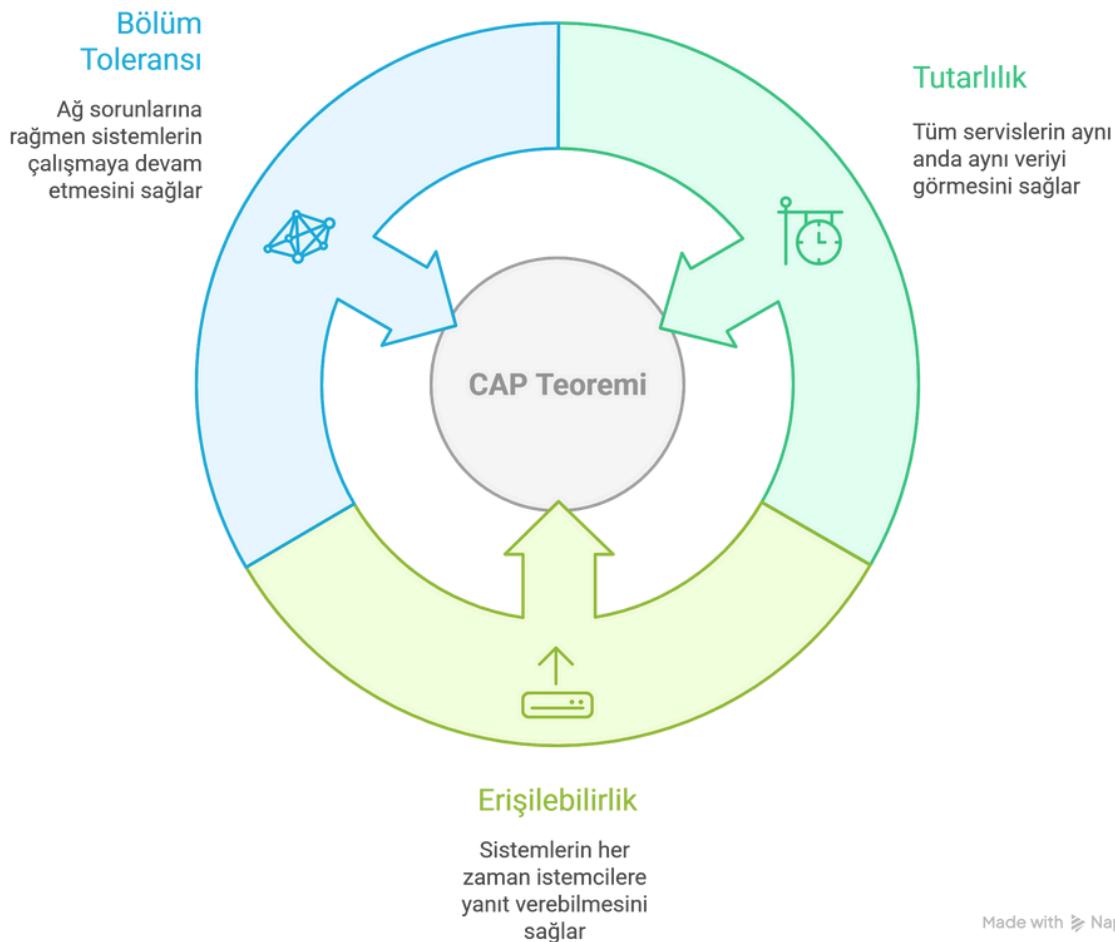
## Olay Tabanlı (Event-Driven) Mimariler

Veri senkronizasyonu için bir diğer yaklaşım, **olay tabanlı (event-driven)** mimaridir. Bu modelde, herhangi bir mikroserviste gerçekleşen bir olay, mesajlaşma altyapısı aracılığıyla diğer servislere asenkron olarak iletilir. İlgili servisler bu olayı dinleyerek gerekli işlemleri gerçekleştirir ve böylece veri güncelliliği servisler arasında otomatik olarak sağlanır.

# CAP Teoremi

**CAP Teoremi**, dağıtıksistemlerin tasarılarında karşılaşılan temel zorlukları açıklayan bir prensiptir. Bu teorem, dağıtık sistemlerde Consistency (Tutarlılık), Availability (Erişilebilirlik) ve Partition Tolerance (Bölüm Toleransı) olmak üzere üç temel özelliğin aynı anda tam olarak sağlanamayacağını öne sürer. Sistem tasarılarında bu üç özellik arasında bir denge kurulması gereklidir.

# CAP Teoremi İlkeleri



## Consistency (Tutarlılık)

Tutarlılık, dağıtıksistemdeki tüm servislerin aynı anda aynı veriyi görmesini ifade eder. Eğer bir sistem tutarlılık garantisine sahipse, herhangi bir veri yazıldığında veya güncellendiğinde bu değişiklik, tüm servislerde anlık veya en kısa sürede yansıtılır. Bu, veri bütünlüğünün korunması açısından kritik öneme sahiptir.

## Availability (Erişilebilirlik)

Erişilebilirlik, sistemin her zaman istemcilere yanıt verebilme yeteneğidir. Yani bir servis hata verse bile, sistem çalışmaya devam eder ve kullanıcıya hizmet sunar. Yüksek erişilebilirlik, özellikle kesintisiz hizmet gerektiren uygulamalar için hayatı bir gereksinimdir.

## Partition Tolerance (Bölüm Toleransı)

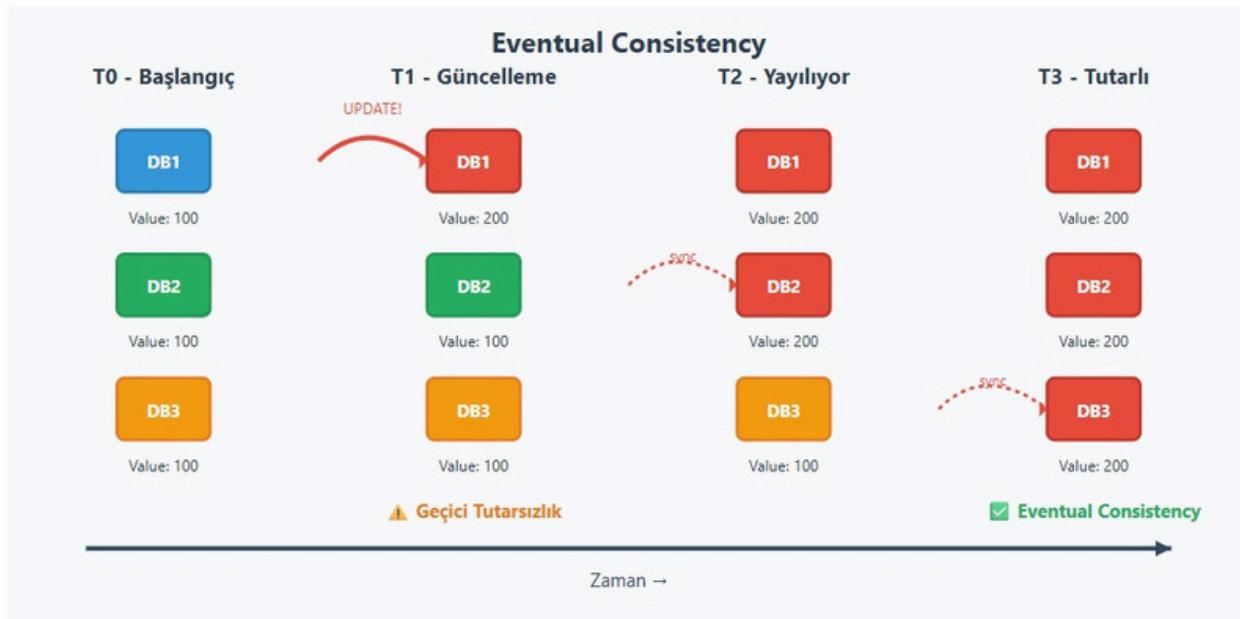
Bölüm toleransı, ağıda paket kaybı, gecikme veya bağlantı kopması gibi iletişim sorunları yaşansa da sistemin çalışmaya devam edebilme yeteneğidir. Ağın tamamen bölünmesi (network partition) durumunda bile, sistem kalan bileşenleriyle işlemeye devam edebilmelidir. CAP Teoremi, bu üç özelliğin (Consistency, Availability, Partition Tolerance) aynı anda eksiksiz olarak sağlanamayacağını belirtir. Teoreme göre, bir dağıtık sistem tasarılanırken mutlaka bu üç ilkeden en az biri tam olarak yerine getirilemez; dolayısıyla sistem ya CA, CP ya da AP özelliklerini birlikte sağlayacak şekilde kurgulanır.

# Eventual Consistency & Strong Consistency

## Eventual Consistency :

**Eventual Consistency** (Nihai Tutarlılık), dağıtık sistemlerde veri tutarlığını ve senkronizasyonunu yönetmek için kullanılan bir yaklaşımdır. Bu modelin amacı, verinin tüm servislerde zamanla tutarlı hale gelmesini sağlamak. Yani, bir serviste yapılan bir veri değişikliği diğer servislere anında yansıtılmaz; bunun yerine, belirli bir süre (genellikle birkaç milisaniye veya saniye) içinde tüm kaynaklarda aynı veri seti elde edilir. Böylece sistem genelinde **nihai olarak** tutarlılık sağlanır.

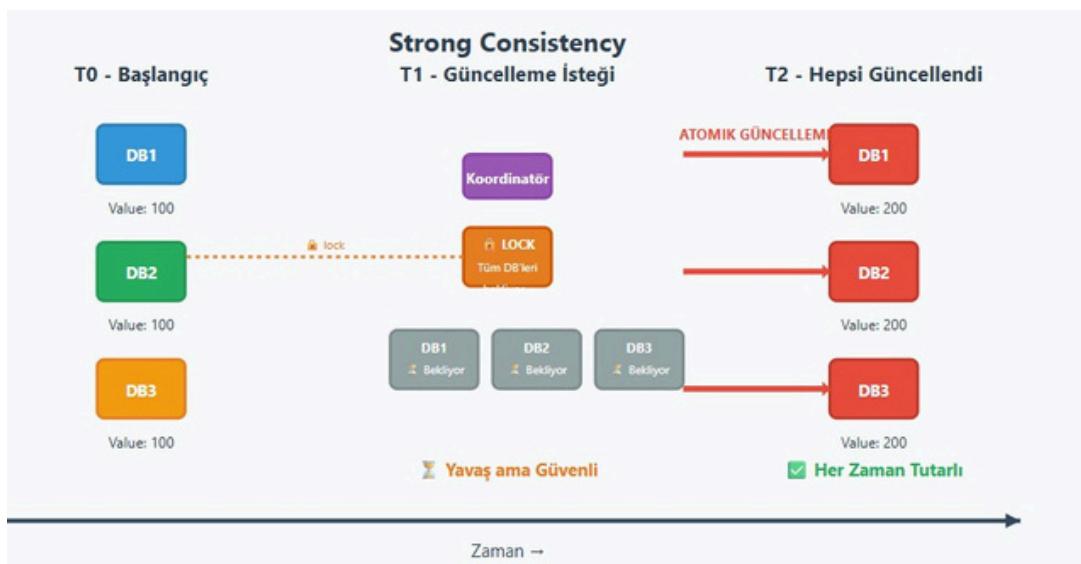
Eventual consistency, sistemin ölçeklenebilirliğini ve servisler arası bağımsızlığı korurken, veri güncellemelerinin hızlı bir şekilde dağıtılmamasına ve işlenmesine olanak tanır. Ancak bu model, geçici olarak tutarlı olmayan verilerle çalışmayı gerektirebilir. Bu durum, bazı senaryolarda **tutarlılık yanılığına** veya **eski verilerin görüntülenmesine** yol açabilir. Yine de, bu tutarsızlıklar çoğunlukla saniyeler seviyesinde olduğundan pratikte göz ardı edilebilir.



Eventual consistency, bir sistemde **tutarlılık**, **erişilebilirlik** ve **bölüm toleransı** arasındaki dengeyi açıklayan **CAP Teoremi** ile doğrudan ilişkili bir stratejidir.

## Strong Consistency :

**Strong Consistency** (Güçlü Tutarlılık), dağıtık sistemlerde tüm servislerde her zaman aynı ve en güncel verinin bulunmasını garanti eden bir yaklaşımdır. Bu modelde, **eventual consistency**'de görülen kısa süreli tutarsızlıklar söz konusu değildir; veri değişikliği yapıldığında tüm servisler anında güncel veriyi yansıtır. Strong consistency, veri bütünlüğü ve tutarlılığının kritik olduğu senaryolarda tercih edilir. Özellikle **finansal işlemler**, **envanter yönetimi** gibi hataya toleransı olmayan alanlarda önemli bir rol oynar. Ancak strong consistency'yi sağlamak, performans ve ölçeklenebilirlik üzerinde etkiler yaratabilir. Bunun nedeni, veri güncellemelerinin tüm servisler arasında koordineli bir şekilde dağıtılması gerekliliğidir. Bu durum, sistem genelinde **yüksek gecikme sürelerine** ve **kısıtlı performans düşüşlerine** yol açabilir. Dolayısıyla, güçlü tutarlılık tercih edildiğinde, sistem tasarımda bu performans-tutarlılık dengesi dikkate alınmalıdır.



## Eventual vs Strong :

Özellik	Eventual Consistency	Strong Consistency
Tutarlılık Garantisi	Gelecekte tutarlı olacak	Her zaman tutarlı
Performans	Yüksek performans	Düşük performans
Gecikme	Düşük gecikme	Yüksek gecikme
Ölçeklenebilirlik	Çok iyi ölçeklenir	Sınırlı ölçeklenme
Karmaşıklık	Basit implementasyon	Karmaşık koordinasyon
Maliyet	Düşük maliyet	Yüksek maliyet

## Gerçek Kullanım Örnekleri :

### Eventual Consistency :

Senaryo	Açıklama	Neden Uygun
Sosyal Medya	Facebook beğeni sayıları, Twitter takipçi sayısı	Anlık tutarlılık kritik değil
E-ticaret Ürün Kataloğu	Amazon ürün açıklamaları, fiyat güncellemeleri	Birkaç saniye gecikme kabul edilebilir
DNS Sistemleri	Domain name çözümleme	Global dağıtım, gecikme tolere edilir
CDN ve Cache	Cloudflare, AWS CloudFront	İçerik dağıtımında gecikme normal
IoT Sensör Verileri	Hava durumu, trafik sensörleri	Gerçek zamanlı olmasa da çalışır
Log ve Analytics	Google Analytics, sistem logları	Raporlama için anlık veri gerekli değil
Email Sistemleri	Gmail, Outlook senkronizasyonu	Birkaç saniye gecikme problem değil

## Strong Consistency :

Senaryo	Açıklama	Neden Zorunlu
Bankacılık İşlemleri	Para transferi, hesap bakiyesi	Tutarsızlık parasal kayıp demek
Borsa İşlemleri	Hisse senedi alım/satım	Anlık tutarlılık kritik
Envanter Yönetimi	Stok kontrolü, rezervasyon	Overselling önlenmeli
Ödeme Sistemleri	Kredi kartı işlemleri	Çift ödeme engellenilmeli
Rezervasyon Sistemleri	Otel odası, uçak biletleri	Aynı kaynak iki kez satılamaz
Kimlik Doğrulama	Login, authorization	Güvenlik kritik
Database Transactions	ACID işlemler	Veri bütünlüğü şart
Voting Sistemleri	Seçim oyları, anket	Her oy sayılmalı

**Eventual consistency**, bir serviste gerçekleşen veri güncellemesinin diğer servislere anında yansıtılmasının gerekmeyiği durumlarda tercih edilir. Buna karşılık, **strong consistency**, bir serviste yapılan veri değişikliğinin diğer tüm servislere anında yansıtılmasının kritik olduğu senaryolarda kullanılır.

## Two Phase Commit (2PC) Protocol

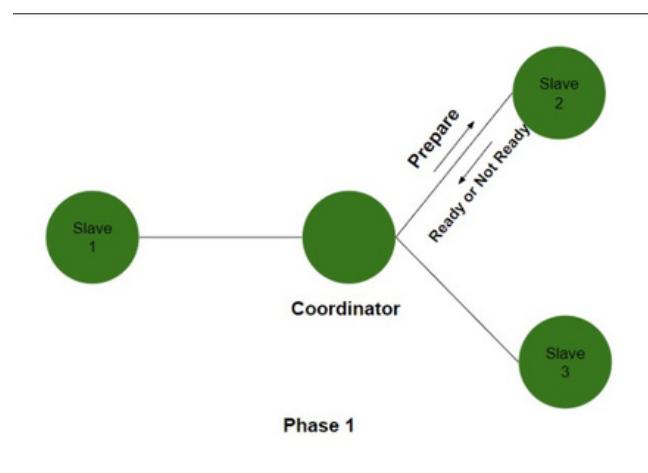
**Two-Phase Commit (2PC)**, dağıtık sistemlerde **strong consistency** modelini destekleyerek veri tutarlığını sağlamak için kullanılan bir protokoldür. Bu protokol, birden fazla servis arasında işlem koordinasyonunu ve veri bütünlüğünü yönetir. Temel amacı, bir işlemin **tüm kaynaklarda atomik** olarak tamamlanmasını garanti etmektir; yani işlem ya tüm katılımcılar tarafından başarıyla uygulanır ya da hiçbirinde uygulanmaz.

### 1. Prepare Phase (Hazırlık Aşaması)

**Koordinatör:** Kullanıcıdan gelen işlem talebini alır ve bu talebe dahil olan tüm **katılımcı** node'lara "hazır olup olmadıkları" konusunda bir **prepare** mesajı gönderir.

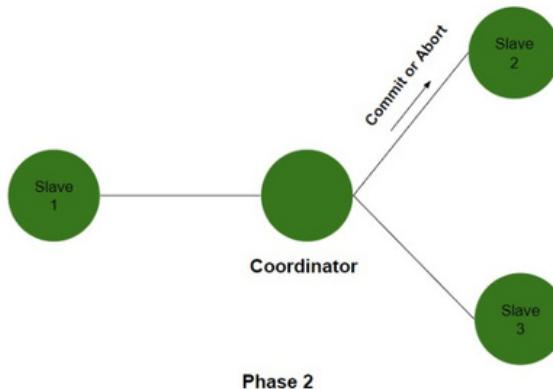
**Katılımcılar:** Gelen isteği değerlendirerek işlemi gerçekleştirmeye hazır olup olmadıklarını belirtir.

Tüm katılımcılardan "**Yes/OK**" yanıtı gelirse, koordinatör **commit aşamasına** geçer. En az bir katılımcı "**No/Abort**" yanıtı verirse ya da belirli bir süre içinde yanıt alınamazsa, koordinatör işlemi iptal eder ve tüm katılımcılara **abort** mesajı gönderir.



## 2. Commit Phase (Onay Aşaması)

**Koordinatör**, tüm katılımcıların "hazır" onayı almasının ardından, işlem başlatılması için commit mesajını ileter. Katılımcılar, işlemi tamamladıktan sonra koordinatöre "Ack/OK" yanıtını gönderir. Tüm katılımcılardan beklenen onay mesajları geldikten sonra işlem başarıyla tamamlanmış kabul edilir. Eğer katılımcılardan herhangi birinden hata mesajı gelirse veya yanıt alınamazsa, koordinatör işlemini iptal eder ve tüm katılımcılara abort mesajı göndererek yapılan değişikliklerin geri alınmasını sağlar.



Görüldüğü üzere **Two-Phase Commit (2PC)** protokolü, veri tutarlığını sağlamak amacıyla bir işlemi ya tüm servislerde uygular ya da hiçbirinde uygulamaz. Özellikle bir katılımcının uzun süre yanıt vermemesi veya hata oluşması durumunda, gerçekleştirilen tüm işlemler geri alınır (**rollback**). Bu durum, yüksek gecikme süreleri ve ek işlem maliyeti nedeniyle 2PC protokolünün performans sorunlarına yol açmasına neden olabilir.

### Koordinatör :

2PC protokolünün merkezi yönetici olan Koordinatör (Coordinator) servisi, tüm işlemin başlatılmasından her microservice'in hazır olup olmama durumuna kadar tüm adımları kontrol etmekte, süreçte beklemekte ve ardından da tüm aşamasındaki onayın, red'in olmak üzere işlemin veri teyit veya abort mesajını göndererek katılımcılara gerekli komutları veren birimdir.

### Katılımcılar :

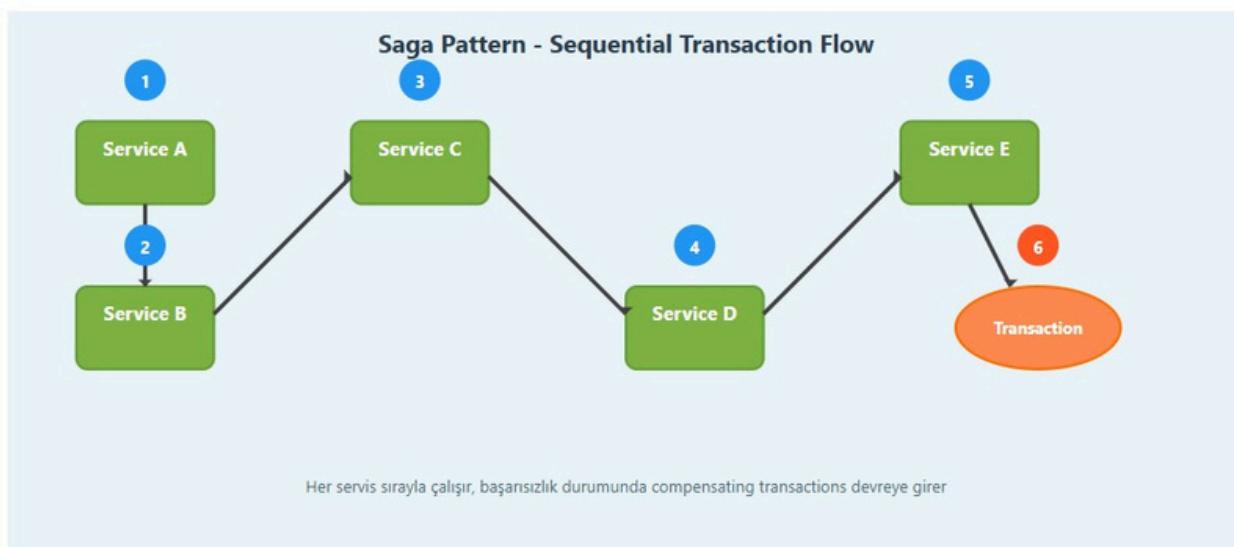
Katılımcılar(Participants), 2PC protokolü sürecinde, kullanıcidan talep doğrultusunda işlemi operasyonla yürütecek olan mikroservislerin ta kendileridir. Talep yapılan operasyonlara göre işlemlerini commit etmekte ve koordinatöre onay verip bilgi vermekte ve koordinatöre göre işlemlerini abort etmeyi beklemektedir. Ayrıca teknik olarak nitelendirilebilir mikroservis, prepare aşamasında görülen koordinatöre cevaplayacaktır.

# Eventual Consistency - Saga Pattern

**SagaPattern** (Saga Modeli), dağıtık sistemlerde ve mikroservis mimarilerinde kullanılan bir distributed transaction yönetim desenidir. Bu desenin temel mantığında, ilk işlem (transaction) genellikle bir dış etkileşim — örneğin bir butona tıklama — ile tetiklenir. Bu işlem başarıyla tamamlandığında, bir sonraki servisteki transaction başlatılır ve süreç bu şekilde, zincirleme olarak diğer servisler üzerinde devam eder. Saga Pattern, herhangi bir adımda hata oluşması durumunda Compensating Transaction adı verilen bir hata işleme stratejisini devreye sokar. Bu strateji, daha önce gerçekleştirilen adımların etkilerini geri alarak (rollback) işlemi iptal eder. Böylece veri tutarlılığı korunur, istemsiz tutarsızlıklar önlenir ve Atomicity prensibi desteklenmiş olur.

## Saga Pattern'in Amacı

Saga Pattern'in temel amacı, dağıtık sistemlerde ve mikroservis mimarilerinde **uzun süreli ve karmaşık işlemleri yönetmek** için güvenilir ve ölçeklenebilir bir çözüm sunmaktır. Bu sayede, merkezi bir kilitleme mekanizması kullanmadan, servisler arası koordinasyon sağlanarak veri bütünlüğü korunur.



Saga Pattern, uzun ömürlü işlemlerin yönetilmesini ve adımlar hâlinde parçalanmasını hedefler. Bu yaklaşım, servislerden herhangi birinde hata meydana gelebileceği ihtimalini göz önünde bulundurarak sistemin esnekliğini artırmayı amaçlar. Dağıtık transaction sürecinde — hatta hata durumlarında dahı — sistemdeki tutarlılığın korunmasını sağlar. İşlemlerin asenkron veya paralel şekilde yönetilmesine olanak tanır. Her bir adımın ayrı bir mikroservis tarafından yürütülmesi sayesinde sistem daha **modüler** ve **ölçeklenebilir** bir yapıya kavuşur. Kısacası Saga Pattern, dağıtık sistemlerdeki karmaşık işlemleri yönetmek ve hata durumlarına karşı koruma sağlamak üzere tasarlanmış bir modeldir.

## Saga Pattern – Eventual Consistency

Saga Pattern, doğası gereği **Eventual Consistency** yaklaşımını benimser. Bu, iş akışı sürecinde farklı adımlar veya servisler arasında kısa süreli gecikmeler olabileceği anlamına gelir. Yani herhangi bir adımda yapılan değişiklikler, tüm adımlara anında yansımayabilir. Ancak iş akışı tamamlandığında sistem genelinde **nihai tutarlılık** sağlanır.

# Saga Pattern'i Uygulama Yaklaşımları

Saga Pattern teknik olarak iki farklı şekilde uygulanabilir:

**Events / Choreography** : Servislerin birbirlerini olaylar (events) aracılığıyla tetiklediği, merkezi koordinasyonun bulunmadığı yaklaşım.

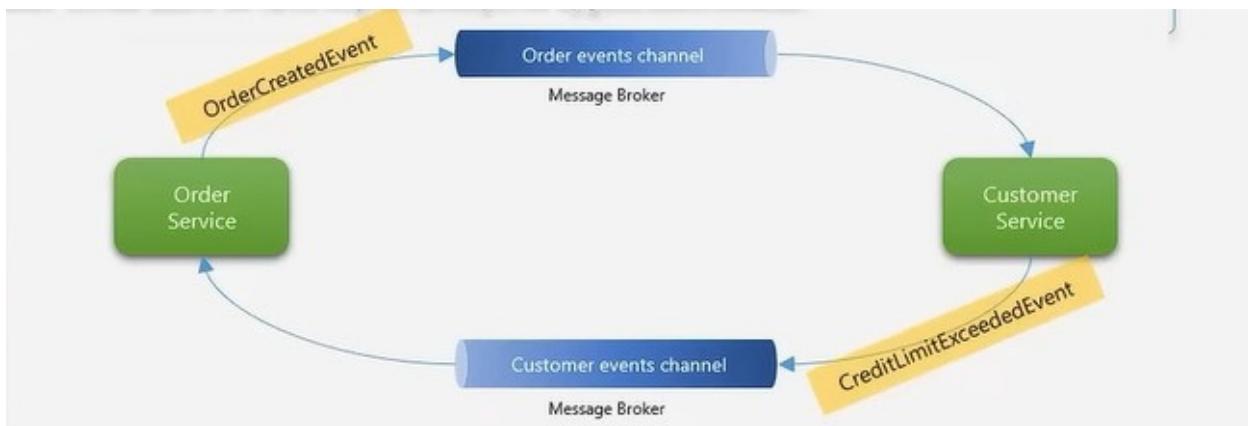
**Command / Orchestration** : Tüm adımların merkezi bir "orchestrator" servis tarafından yönetildiği yaklaşım

## Saga Pattern - Choreography

Bu yaklaşımın mikroservisler arasında merkezi bir denetim veya kontrol noktası bulunmaz. Servisler, birbirlerini doğrudan tetiklemek yerine message broker üzerinden iletilen olaylar (events) aracılığıyla haberleşir. Taktiksel olarak süreç şu şekilde işler:

- Transaction ilk serviste başlar.
- Bu servis işlevini tamamladıktan sonra, bir sonraki servisi tetiklemek amacıyla message broker üzerinden bir event yayarlar.
- Ardından, tetiklenen servis de kendi işlemini tamamlayarak yine bir event yayarlar ve zincir bu şekilde devam eder.
- İşlem, istemcinin talep ettiği sürecin son servise ulaşmasıyla tamamlanır.

Her bir servis, kendisinden sonraki adımın başlatılıp başlatılmayacağına kendi karar verir. Servis, işlem sonucunda **başarılı** veya **başarısız** durumunu değerlendirir; başarılı ise bir sonraki servisi tetikler, başarısız ise tüm sürecin geri alınması (compensating transactions) başlatılır. Bu nedenle **Choreography** modelinde her servis kendi başına bir "karar verici" konumundadır. Choreography yaklaşımı, genellikle dağıtık transaction sürecine katılan servis sayısının **2 ile 4** arasında olduğu durumlarda tercih edilir. Eğer 4'ten fazla servis süreçte dahil olacaksa **Orchestration** yaklaşımı daha uygun bir çözüm olur. Choreography'de her servis, ilgili kuyruğu (queue) dinler ve tanımlı event türüne uygun bir mesaj geldiğinde gerekli işlemleri gerçekleştirir. İşlemenin sonucuna göre "başarılı" veya "başarısız" bilgisi üretilir. Diğer servisler bu sonuca göre ya kendi işlemlerine devam eder ya da tüm transaction süreci geri alınarak **veri tutarlılığı** korunur.

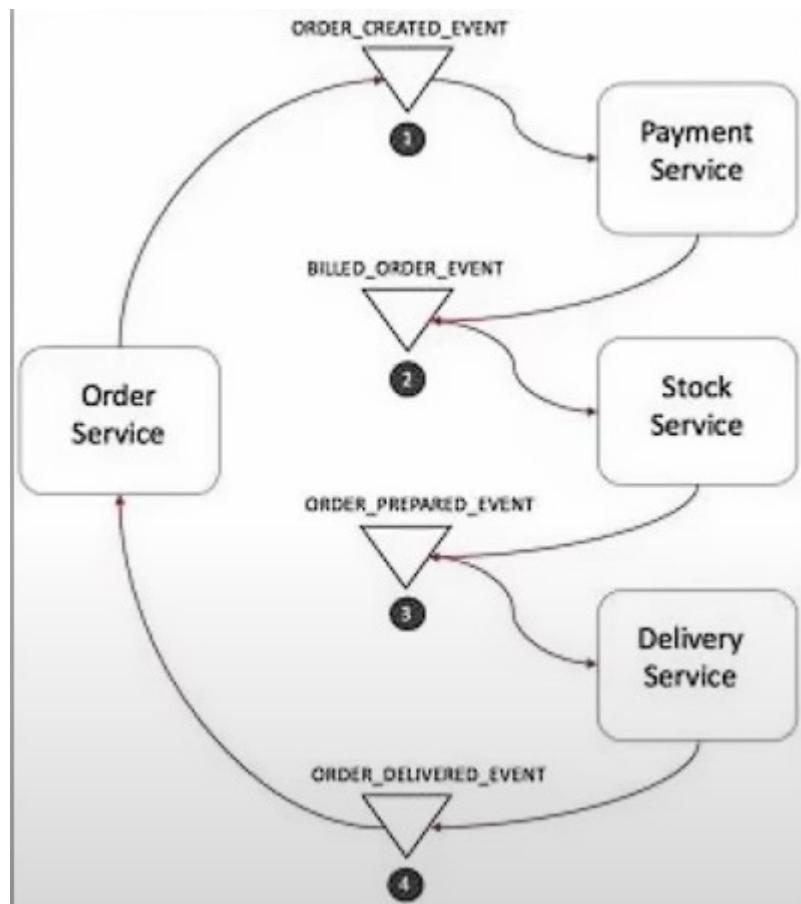


Bu uygulamada bir siparişi oluşturabilmek için;

- Order Service'te alınan POST isteği neticesinde sipariş oluşturulur.
- Order events channel kuyruğuna 'OrderCreatedEvent' isimli event gönderilir.
- Customer Servisi ise Order events channel'da ki 'OrderCreatedEvent'a subscribe'tır. Haliyle ilgili kuyruğa beklenen türde event geldiğinde Customer Service tetiklenmektir.
- Customer Service, müşteriye dair gerekli tüm işlemleri yaptıktan sonra eğer işlemler başarılıysa 'OrderSucceededEvent'i, yok eğer başarısız değilse 'CreditLimitExceededEvent' isimli event'i Customer events channel kuyruğuna gönderecektir.
- Haliyle ilgili kuyruğa subscribe olan Order Service gelen event'in türüne göre ya siparişi onaylayacak ya da reddedecektir.

Göründüğü üzere servisler arası uçtan uca bir iletişim olmadığı için coupling azalacaktır. Ayrıca transaction yönetimi olmadığı için performance bottleneck azalacaktır. Ve dikkat ederseniz eğer choreography yöntemi sayesinde aorumluluklar Saga katılımcı servisleri arasında dağıtılmışından dolayı tek bir hata noktası olmayacağıdır. Dolayısıyla bu durumdan dolayı ekstra bakım gerekmemektedir.

Şimdi aşağıdaki şema üzerinden gerçek bir senaryoyu ele alalım :

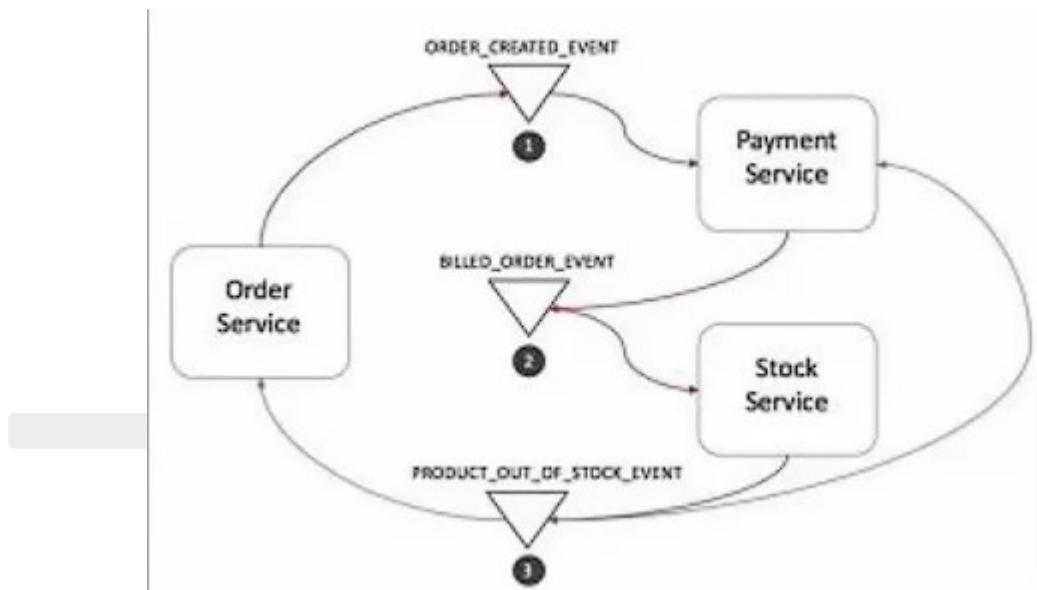


Bu uygulama senaryosunda bir siparişin oluşturulma süreci şu adımlarla ilerler:

- **Order Service**, gelen **POST** isteği ile yeni bir sipariş oluşturur. Sipariş oluşturulduktan sonra **Order Events Channel** kuyruğuna OrderCreatedEvent adlı bir event gönderilir.
- **Customer Service**, **Order Events Channel** üzerinde **OrderCreatedEvent** türüne abonedir (subscribe). Kuyruğa bu event geldiğinde otomatik olarak tetiklenir.
- **Customer Service**, müşteriyle ilgili gerekli tüm kontrolleri ve işlemleri gerçekleştirir: İşlem başarılıysa, Customer Events Channel kuyruğuna OrderSucceededEvent event'ini gönderir. İşlem başarısızsa (örneğin kredi limiti yetersizse), Customer Events Channel kuyruğuna CreditLimitExceededEvent event'ini gönderir.

### Avantajlar:

Servisler arasında doğrudan ucta iletişim olmadığından **coupling** azalır. Merkezi transaction yönetimi bulunmadığından **performans darboğazı** (performance bottleneck) riski düşer. Choreography yöntemi sayesinde sorumluluklar Saga katılımcı servisleri arasında dağıtilır; dolayısıyla tek bir hata noktası (**single point of failure**) oluşmaz. Bu yaklaşım ek bakım gereksinimini azaltır.



**Adım 1 :** Ödeme işlemi tamamlandıktan sonra **Stock Service** üzerinde stok kontrolü yapılır. Eğer yeterli stok bulunmadığı tespit edilirse **PRODUCT\_OUT\_OF\_STOCK\_EVENT** isimli event yayınlanır (publish edilir)

**Adım 2:** **PRODUCT\_OUT\_OF\_STOCK\_EVENT** event'ine abone (subscribe) olan Payment Service ve

**Order Service**, daha önce gerçekleştirilmiş işlemleri geri alır (**compensating transactions**). Bu ters işlemler şunlardır:

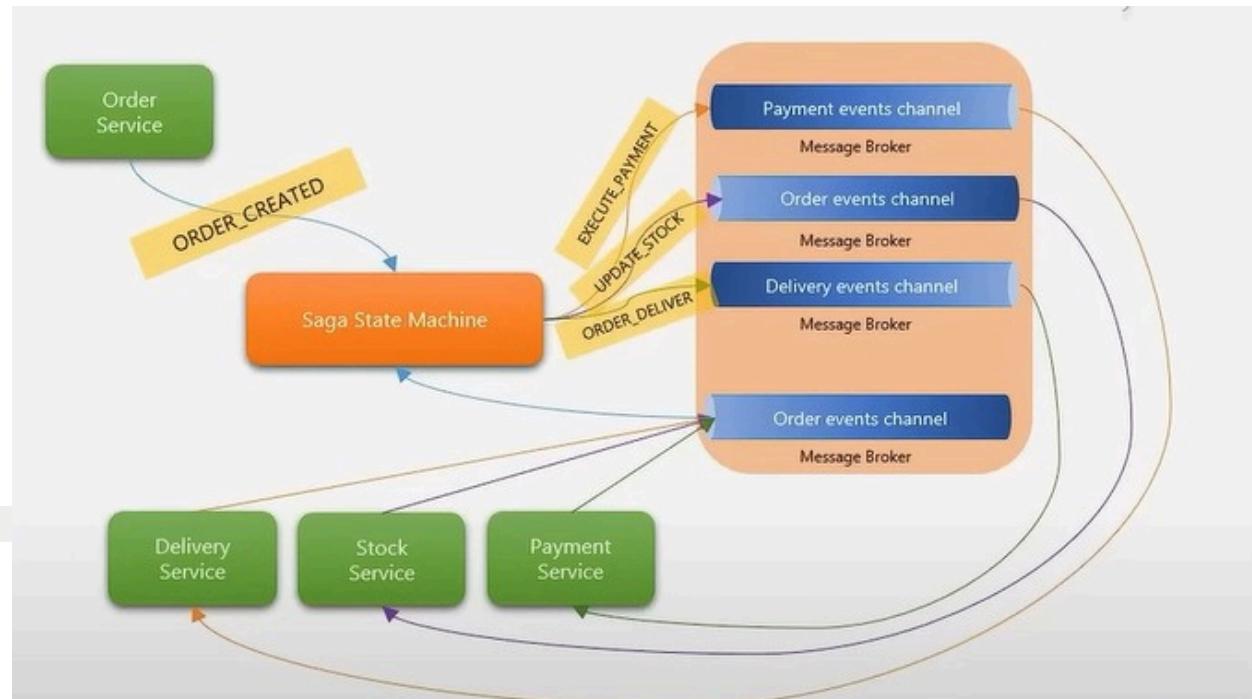
- **Payment Service:** Kullanıcıdan alınan ödeme iade edilir.
- **Order Service:** Sipariş durumu "**Fail**" olarak güncellenir.

## Events / Choreography – Dezavantajlar

- **Servis-Abonelik Takibi Zorluğu:** Hangi servisin hangi kuyruğu dinlediğini (subscribe olduğu event'i) takip etmek zamanla zorlaşır. Bu durum, iş akışının anlaşılması ve yönetilmesini güçleştirir.
- **Döngüsel Bağımlılık Riski:** Servisler birbirlerinin kuyruklarını tükettiğinde, sistemde döngüsel bağımlılıklar ortaya çıkabilir.
- **Zor Entegrasyon Testi:** Bir iş akışını uçtan uca simülle edebilmek için tüm ilgili servislerin çalışır durumda olması gereklidir; bu da entegrasyon testlerini karmaşık hale getirir.

## Saga Pattern - Orchestration

Bu yaklaşımın, servisler arası distributed transaction süreci merkezi bir denetleyici aracılığıyla koordine edilir. Bu merkezi denetleyiciye Saga Orchestrator ya da diğer adıyla Saga State Machine denir. Saga Orchestrator, servisler arasındaki tüm adımları yönetir ve her adımın sonucuna göre hangi işlemin başlatılacağını belirler. Diğer adı olan Saga State Machine ifadesinden de anlaşılacağı üzere, bu bileşen her kullanıcı isteğine ait uygulama state bilgisini tutar, yorumlar ve sürecin ilerleyişini takip eder.



**Adım 1 :** Order Service, sipariş isteğini alır ve sipariş durumunu Suspend olarak kaydeder. Ardından, siparişe ait geri kalan işlemleri başlatmak üzere Saga State Machine'e ORDER\_CREATED komutunu gönderir. Böylece sipariş oluşturma transaction'ı başlatılmış olur.

**Adım 2 :** Saga State Machine sırasıyla ; EXECUTE\_PAYMENT komutunu Payment Service'e, UPDATE\_STOCK komutunu Stock Service'e, ORDER\_DELIVER komutunu Delivery Service'e gönderir.

Bu komutları alan servisler gerekli işlemleri yürütür ve sonuçlarını (başarılı veya başarısız) geri bildirir. Eğer tüm adımlar başarıyla tamamlanırsa sipariş durumu **Completed** olarak güncellenir. **Hata Durumu Örneği:** Senaryodaki servislerden herhangi birinde hata meydana geldiğinde **rollback** süreci başlatılır. Örneğin, **Stock Service** üzerinde stok yetersizliği oluşursa:

- **Stock Service**, sipariş edilen ürün adedinin mevcut stok miktarından fazla olduğunu tespit eder ve State Machine'e **OUT\_OF\_STOCK** komutunu gönderir.
- **State Machine**, bu bilgi üzerine işlemin başarısız olduğunu kaydeder ve ilgili rollback adımlarını tetikler.

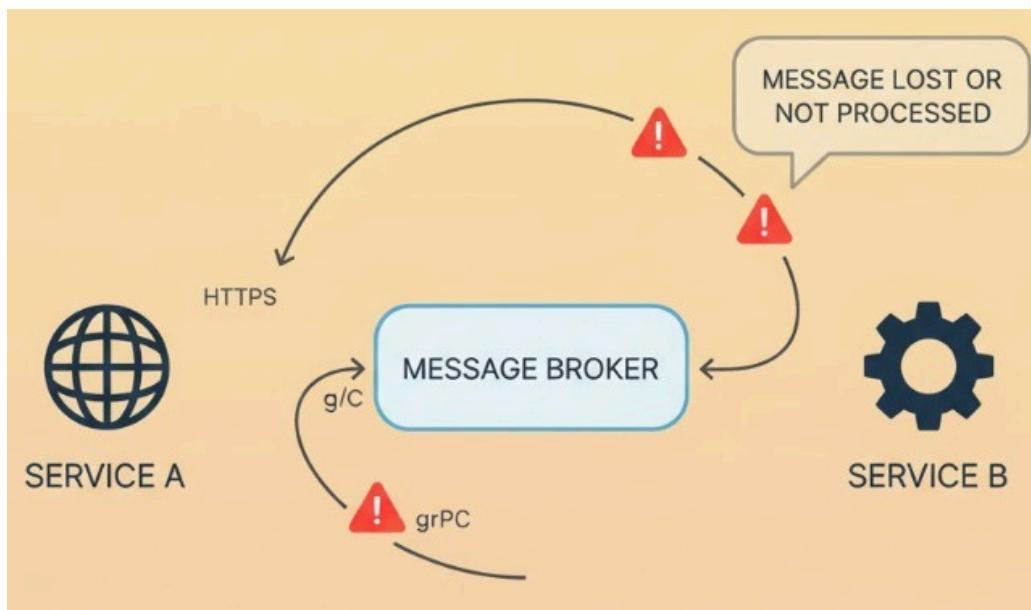
**Not:** Her işlem için **State Machine** üzerinde state bilgisinin tutulması, sürecin adım adım izlenmesini sağlar. Bu sayede hem sürecin hangi aşamada olduğu hem de hata durumlarında hangi noktada yanlış yönetim yapıldığı kolayca tespit edilebilir.

## Orchestration Yaklaşımının Faydaları Nelerdir :

Birçok servisin bulunduğu ve zaman içinde servislerin eklediği karmaşık iş akışları için idealdir. Her bir servisin bu servis faaliyetlerinin üzerinde merkezi bir kontrol sağlar. Orchestration implementasyonu, tek taraflı olarak servislere bağlı olduğundan daha kolay geliştirilebilir. Her bir servisin diğer servislerle ilgili bilmesi gereken herhangi bir bilgi yoktur.

## Servisler Arası Mesaj Güvenliği-Outbox ve Inbox Desing Pattern:

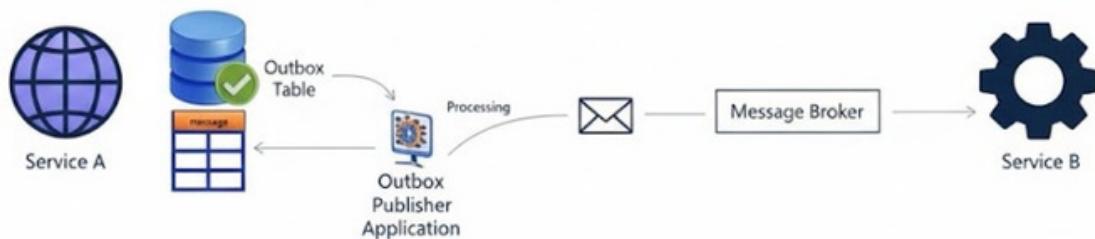
Gönderilen mesajların hedefine ulaşması ve doğru bir şekilde işlenmesi, teknolojik sistemlerin en önemli sorunlarından biridir. Bir **A Servisi**'nden **B Servisi**'ne bir mesaj gönderdiğinizde, bu mesajın bağlantı kopukluğu gibi nedenlerle yolda kaybolması veya B Servisi tarafından tam olarak işlenmemesi gibi sorunlar ortaya çıkabilir. Bu durum, mesajlaşma için hangi yöntemin ya da protokolün kullanıldığı fark etmeksizin meydana gelebilir. Bu tür sorunları aşmak için çeşitli mekanizmalar ve aracı sistemler kullanılır. Örneğin, mesajların doğrudan alıcıya gönderilmesi yerine bir **Mesaj Aracı Kurumu (Message Broker)** üzerinden iletilmesi tercih edilebilir. Bu sistemler, mesajların güvenli bir şekilde depolanmasını ve alıcının müsait olduğunda mesajı almasını sağlar. Ancak, en güvenilir yöntemler bile tam bir garantisini vermez ve mesajın yolda kaybolma veya hedefine ulaştıktan sonra işleme esnasında sorun yaşama ihtimali her zaman bulunur. Bu, sistemlerin dayanıklılığını ve hata toleransını artıran ek önlemler almayı gerektirir.



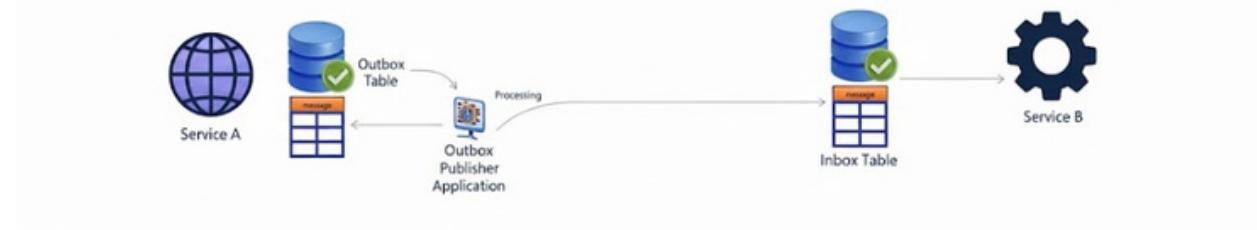
Mesajın güvenliğini kim sağlayacaktır?

Bir servis, ilk adımda mesajını başka bir servise göndermeden önce bir veritabanına kaydederek, mesajın gönderildiğine dair fiziksel bir kimlik oluşturur. Böylece mesajın güvenliği için önemli bir önlem alınmış olur ve mesaj kaybolsa bile erişilebilir kılınır. Devamında ise bu tablodaki her bir satır için işlem yapıp hedef servise mesajları gönderecek bir uygulama devreye girer. Bu uygulama sayesinde mesajlar hedef servise garanti bir şekilde iletilir ve işlenmeye devam eder. Bu uygulamaya **Mesaj Yayımlayıcı Uygulama (Message Publisher Application)** denir. Bu mesajlar gönderildikten sonra silinebilir ancak metrik ve bir nevi log tutma amacıyla veritabanında saklanmaya devam edebilir. Bu durumda, mesajlar "gönderildi" olarak işaretlenir.

Burada anlatılan mesaj güvenliğini sağlayan bu tasarıma **Outbox Tasarım Deseni (Outbox Design Pattern)** adı verilir. Gönderici tarafından mesajların yazıldığı tabloya ise **Outbox Tablosu (Outbox Table)** denir. Bu tablodan mesajları okuyup mesajları yayan uygulamaya da **Outbox Yayımlayıcı Uygulaması (Outbox Publisher Application)** denir.



Gönderici tarafından mesaj güvenliğini, yukarıda anladığımız **Outbox Design Pattern** ile sağlamış olduk. Peki, alıcı tarafından mesaj güvenliğini nasıl sağlayacağız? **Publisher Application** tarafından yayılan mesajlar bir **Inbox Table**'a yazılmasına başlanır ve mesajların iletiliği servis, bu tablo üzerinden mesajları okur. İşte bu yaklaşımı **Inbox Design Pattern** adı verilir.



Bu metinler, dağıtık sistemlerde **Outbox Deseni (Outbox Pattern)** ve **Inbox Deseni (Inbox Pattern)** adı verilen tasarım yaklaşımlarını açıklıyor. Temel amaç, mesajların güvenilir bir şekilde iletilmesini sağlamak ve olası veri kayıplarının önüne geçmektir.

- **Outbox Deseni (Gönderici Tarafı):** Bir servis, mesajı doğrudan hedef servise göndermek yerine, önce kendi veritabanında yer alan bir Outbox Tablosu'na kaydeder. Bu sayede mesajın fiziksel bir kopyası oluşur ve sistemde bir hata olsa bile mesajın gönderildiği bilgisi güvence altına alınır. Daha sonra Outbox
- **Yayımlayıcı Uygulaması (Outbox Publisher Application)** adı verilen ayrı bir uygulama, bu tablodaki mesajları okuyarak hedef servise iletmek üzere bir mesaj aracısına (message broker) gönderir. Bu yaklaşım, ağ hataları veya geçici kesintiler nedeniyle mesaj kaybını önler ve işlemin atomik (büütünsel) olmasını sağlar.
- **Inbox Deseni (Alıcı Tarafı):** Mesajın alıcı tarafında ise, hedef servis gelen mesajı yine kendi veritabanındaki **Inbox Tablosu**'na kaydeder. Bu tablo, mesajın başarıyla alındığının bir kanıtı gibidir. Bu sayede, aynı mesajın birden fazla kez alınması (duplicate mesajlar) gibi sorunların önüne geçilir ve mesajın doğru bir şekilde işlendiği garantiplenir.

## Outbox Design Pattern Hangi Durumlarda Kullanılır :

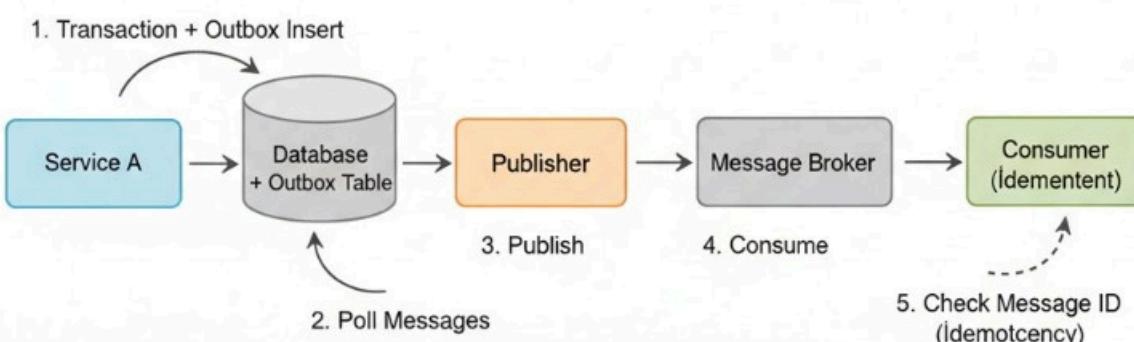
Bu metinler, dağıtık sistemlerdeki mesajlaşma güvenilirliğini artırmak için kullanılan **Outbox Tasarım Deseni**'nin temel prensiplerini açıklıyor. Bu desenin ana fikri, bir servisin hedef servise göndereceği mesajı doğrudan iletmemek yerine, önce kendi veritabanına kaydetmesidir. Bu yaklaşım, mesajın fiziksel olarak bir kopyasının oluşturulmasını sağlayarak, mesajın yolda kaybolması veya ağ sorunları nedeniyle hedefine ulaşamaması gibi riskleri ortadan kaldırır. Böylece, "**çift yazma**" (**dual write**) olarak bilinen ve veritabanına yazma ile mesaj gönderme işlemlerinin atomik olarak (ya hep ya hiç) gerçekleştirilememesi sorununa etkili bir çözüm sunar.

Outbox deseni sayesinde, veritabanına kaydedilen mesajlar, **Outbox Yaymayııcı Uygulaması** tarafından okunarak bir mesaj aracısına gönderilir. Bu süreç, sistemin dayanıklılığını artırır ve mesajların güvenilir bir şekilde hedefe ulaşmasını garanti altına alır. Mesajların veritabanında "gönderildi" olarak işaretlenmesi, hem bir log işlevi görür hem de sistemin durumunu izlenebilir kılar. Bu tasarım, karmaşık telafi edilebilir işlemlerden kaçınarak, dağıtık sistemlerin tutarlılığını sağlamada pratik ve minimalist bir çözüm sunar.

## Outbox Patternde Mesajlar Nasıl Publish Edilmelidir :

Pulling publisher : Outbox tablosunda bulunan mesajların belirli zaman aralıklarında sorgulanarak yayınlanması (publish) işlemini gerçekleştiren bir uygulama geliştirilmesi, Pulling Publisher yöntemi olarak adlandırılır. Bu uygulama, basit bir console application veya worker service şeklinde tasarılanabilir. Pulling publisher uygulamasının Outbox tablosunu sorgulama zaman aralığı ne kadar kısa tutulursa, veritabanına olan maliyet o oranda artar ve mesajlar daha hızlı iletılır. Bu performans-maliyet dengesi, pull yönteminin temel dezavantajını oluşturmaktadır. Transaction log tailing Outbox table'in bulunduğu veritabanının transaction loglarını okuyarak değişiklikleri yakalayıp mesajları publish etme yöntemidir.

## Idempotents Sorunsalı :



Outbox pattern uygulamalarında, mesajlar başarıyla publish edildikten sonra Outbox table üzerinde işaretleme yapılmırken veritabanı ile iletişim hatası yaşanabilir. Bu durumda mesaj başarıyla yayınlanmış olmasına rağmen, Outbox table'a bu durum yansıtılamaz ve sistem tekrar veritabanına bağlandığında aynı mesaj yeniden işlenir. Sonuç olarak duplicate (tekrarlanan) mesajlar oluşur ve bu durum sistemin tutarlılığını olumsuz etkileyebilir. Bu problemi önlemek için consumer servislerinin idempotent şekilde tasarlanması kritik önem taşır. Idempotency, bir mesajın birden fazla kez işlenmesinin sonucu değiştirmemesini garanti eden bir tasarım prensibidir. Bunun için publish edilecek her mesaj veya event'e benzersiz bir kimlik (unique identifier) atanmalıdır. Consumer'lar bu kimlik bilgisini kullanarak mesajın daha önce işlenip işlenmediğini kontrol eder ve duplicate işlemleri engeller. Bu sayede bir mesaj kaç kez gönderilirse gönderilisin, sistemde tek bir işlem olarak ele alınması garantiilenir ve dağıtık sistemlerde mesaj güvenilirliği ile veri tutarlılığı sağlanmış olur.

# Idempotent Sorunsalı

İdempotent terimler, bir işlemin tekrarlı uygulanması durumunda sonucun değişmeden kalmasını ifade eden matematiksel ve bilgisayar bilimi kavramlarıdır. Bir isteğin ilk çağrımda ürettiği sonuç ile sonraki çağrımlarda ürettiği sonuç arasında herhangi bir farklılık oluşmaması, sistemin tutarlığını ve öngörülebilirliğini sağlar. Bu özellik, özellikle dağıtık sistemlerde ve ağ iletişiminde kritik öneme sahiptir çünkü aynı işlemin kasitsız olarak birden fazla kez tetiklenmesi durumunda sistem bütünlüğünü korur ve beklenmeyen yan etkilerin önüne geçer. Mikroservis mimarilerinde idempotent operasyonların tasarımları, güvenilir ve dayanıklı sistemler oluşturmanın temel prensiplerindendir. Özellikle mesaj gönderimi sürecinde kritik rol oynayan bu yaklaşım, ağ kesintileri veya zaman aşımı senaryolarında aynı mesajın birden fazla kez uygulanması durumunda bile tutarlı sonuçlar üretilmesini garanti eder. Benzer şekilde, mikroservisler arasındaki iletişimde güvenlik ve tekrarlanabilirlik gereklilikleri, her servisin idempotent davranış sergilemesini zorunlu kılar. Bu sayede, üçüncü taraf mikroservislerle yapılan entegrasyonlarda dahi sistem genelinde veri tutarlığını korunur ve hata toleransı artırılmış olur. Idempotent operasyonların doğru tasarımları, modern bulut tabanlı ve dağıtık mimarilerde güvenilirlik ve ölçeklenebilirliğin temel yapı taşlarından birini oluşturur.

## İdempotent İlkesinin Kullanılmadığı Durumlardaki Riskler :

İdempotent olmayan bir sistem tasarımda, aynı isteğin tekrar gönderilmesi durumunda sistemdeki durum bilgisi (state) her seferinde değişikliğe uğrar. Bu davranışın arzu edilir veya edilmez olması, tamamen iş gerekliliklerine ve uygulama senaryosuna bağlıdır. Konunun daha iyi anlaşılabilmesi için farklı sektörlerdeki somut örnekler üzerinden değerlendirme yapmak faydalı olacaktır. Para transferi işlemleri, idempotent olmayan operasyonların klasik bir örneğini teşkil eder. Bir transfer işleminin tekrar gönderilmesi halinde hedef hesaptaki bakiye her seferinde gönderilen tutar kadar artış gösterir ve bu durum finansal sistemlerin doğası gereği beklenen ve olması gereken bir davranıştır. Her işlemin benzersiz bir finansal etki yaratması gereğinden, bu senaryoda idempotency istenmeyen bir özelliktir. Veritabanı güncelleme işlemleri ise genellikle idempotent karakterdedir. Örneğin, bir personelin adının "Hilmi"den "Serkan"'a güncellenmesi durumunda, aynı güncelleme isteğinin tekrar edilmesi sonucu değiştirmez çünkü hedeflenen durum zaten elde edilmiştir. E-ticaret sistemlerindeki stok yönetimi operasyonlarında idempotency kritik bir gereklilik olarak önem çıkar. Belirli bir siparişe ilişkin stok düşüm işlemlerinin idempotent şekilde tasarlanması, ağ hatalarından veya sistem yeniden denemelerinden kaynaklı stok tutarsızlıklarının önüne geçer ve envanter bütünlüğünü korur. Benzer şekilde, dosya yükleme süreçlerinde de idempotent yaklaşım benimsenmesi büyük önem taşır. Özellikle büyük boyutlu dosyaların işlendiği senaryolarda, bağlantı kesintileri veya zaman aşımı durumlarda aynı dosyanın birden fazla kez yüklenmesini engellemek için idempotent mekanizmaların uygulanması sistem performansı ve veri tutarlığını açısından elzemdir.

## İdempotent Nasıl Sağlanır :

**Benzersiz İşlem Kimlikleri (Unique Transaction Identifiers)** Her işlem, mesaj veya istekte benzersiz bir kimlik veya anahtar atanması, aynı işlemin tekrar edilip edilmemişinin izlenebilmesi için temel bir mekanizma oluşturur. Bu yaklaşım sayesinde sistem, daha önce işlenmiş bir isteğin yeniden gönderilmesi durumunda bunu tespit edebilir ve gereksiz işlem tekrarlarının önüne geçebilir. Benzersiz tanımlayıcılar genellikle UUID, GUID veya işlem bazlı hash değerleri gibi çakışma olasılığı son derece düşük formatlar kullanılarak üretilir ve her isteğin yaşam döngüsü boyunca takip edilebilirliğini sağlar. Bu yöntem, özellikle dağıtık sistemlerde ve mikroservis mimarilerinde kritik öneme sahiptir çünkü ağ gecikmelerinden veya yeniden deneme mekanizmalarından kaynaklanan çift işlem riskini minimize eder. İşlem kimliklerinin merkezi veya dağıtık bir veri deposunda saklanması ve her yeni istek geldiğinde bu depoya sorgu yapılması, sistemin idempotent davranış sergilemesini garanti altına alır. Böylece aynı işlemin birden fazla kez uygulanması engellenerek veri tutarlığı ve sistem güvenilirliği korunmuş olur.

## Atomik İşlemler (Atomic Transactions)

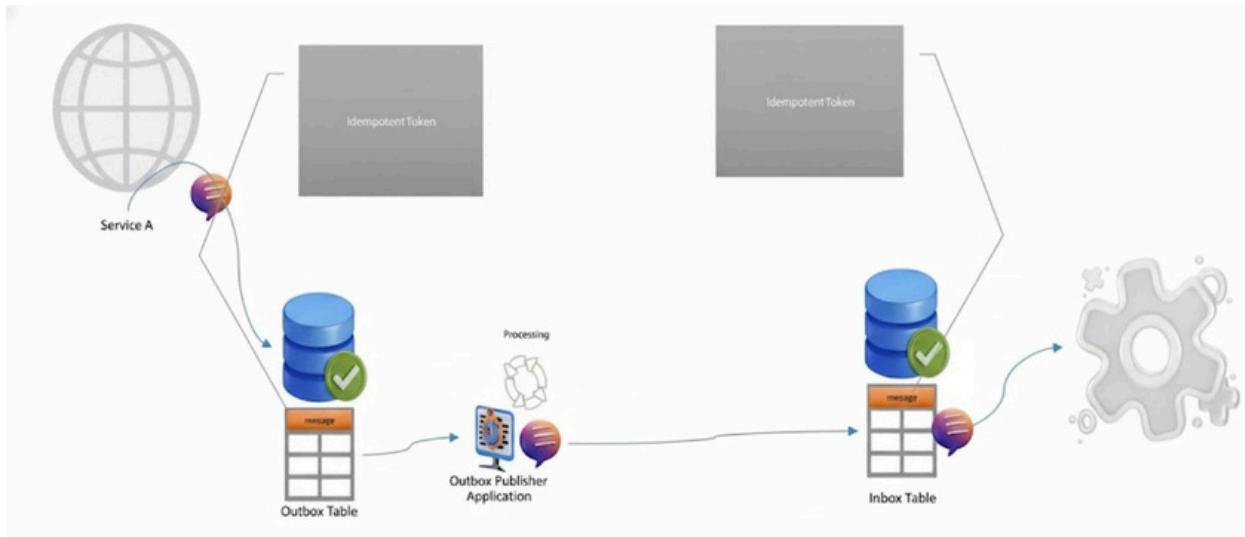
İşlemlerin idempotent garantisi sağlayabilmek için, sistemin atomik hale getirilmesi veya tamamlanmayan işlemlerin geri alınabilmesi (rollback) mekanizmalarıyla desteklenmesi gerekmektedir. Bu davranış, işlemlerin tekrar etme veya yarı kalma durumlarını yaşamadan, minimum süre içinde başarıyla tamamlanmasını veya tamamen geri alınmasını temin eder. Atomik işlemler "ya hep ya hiç" (all-or-nothing) prensibine dayanır ve bir işlem dizisindeki tüm adımların başarılı olması durumunda değişikliklerin kalıcı hale gelmesini, herhangi bir adımda hata oluşması durumunda ise sistemin başlangıç durumuna dönmesini sağlar. Bu yaklaşım, özellikle finansal sistemler, sipariş yönetimi ve envanter güncellemeleri gibi kritik iş süreçlerinde vazgeçilmezdir. Transaksiyonel bütünlüğün korunması, veri tutarsızlıklarını öner ve sistem genelinde güvenilir bir işlem akışı garanti eder.

## İşlem Sonuçlarını Cache'leme (Caching of Transaction Results)

İşlemlerin sonuçlarını önbellekte olarak saklamak, aynı işlemin tekrar edilmesinin gerektiği durumlarda öncelikli bir doğrulama mekanizması sunar. Bu sayede, daha önce başarıyla tamamlanmış bir işleme ait sonuç cache'den hızlıca okunabilir ve işlemin yeniden yürütülmesine gerek kalmadan kullanıcıya aynı yanıt döndürülebilir. Bu yaklaşım hem performans optimizasyonu hem de idempotency garantisi açısından çift yönlü fayda sağlar. Önbellekleme stratejisi, işlem sonuçlarının belirli bir süre boyunca saklanması ve aynı benzersiz tanımlayıcıya sahip isteklerin geldiğinde cache'deki sonuçların kullanılmasını içerir. Bu yöntem özellikle yüksek trafikli sistemlerde veritabanı yükünü azaltır, yanıt sürelerini iyileştirir ve gereksiz hesaplama maliyetlerini ortadan kaldırır. Ayrıca, geçici ağı sorunları veya istemci tarafı yeniden denemeler nedeniyle aynı isteğin birden fazla gönderilmesi durumunda sistemin tutarlı davranışını korur ve idempotent işlem garantisini güçlendirir.

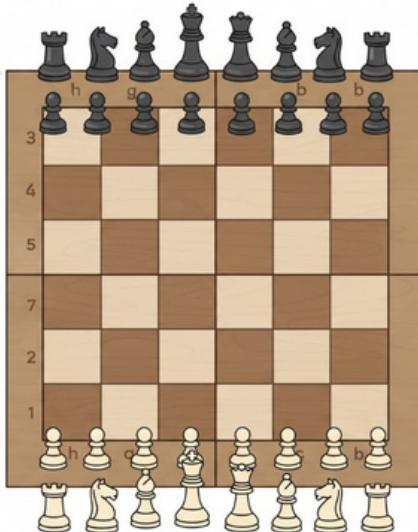
## Zaman Damgası (Timestamp)

Her işlem, mesaj veya istekte zaman damgası ekleyerek, sistemin aynı işlemin tekrar edilip edilmediğini kontrol edebilme kapasitesi kazanması sağlanır. Zaman damgaları, işlemlerin kronolojik sıralamasını belirlemek ve belirli bir zaman penceresi içinde tekrarlanan istekleri filtrelemek için kritik bir rol oynar. Bu mekanizma özellikle eşzamanlılık sorunlarının yönetiminde ve sıralama gerekliliklerinin olduğu sistemlerde vazgeçilmezdir. Zaman damgası bazı idempotency kontrolü, genellikle benzersiz işlem kimlikleriyle birlikte kullanılarak daha güçlü bir koruma katmanı oluşturur. Sistem, belirli bir zaman aralığı içinde aynı işlem kimliğine sahip birden fazla isteği tespit ettiğinde, bunları yinelenen istekler olarak değerlendirir ve sadece ilk isteği işler. Bu yaklaşım ayrıca eski isteklerin temizlenmesi ve cache yönetimi için de referans noktası sağlar, böylece sistem kaynaklarının verimli kullanımı optimize edilmiş olur.

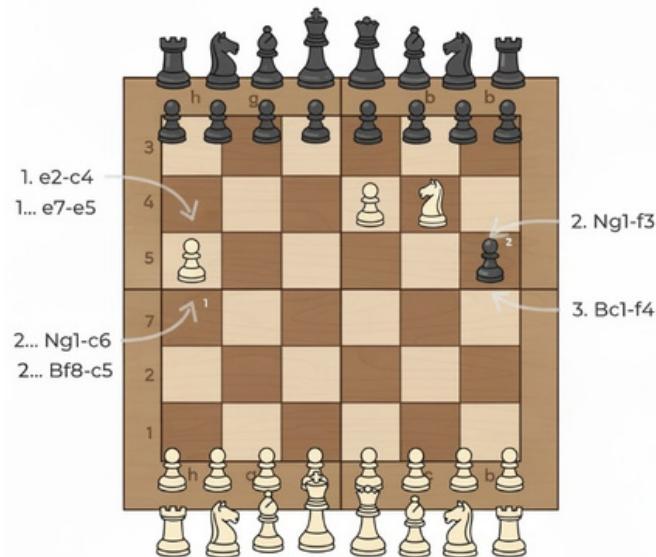


# Event Sourcing

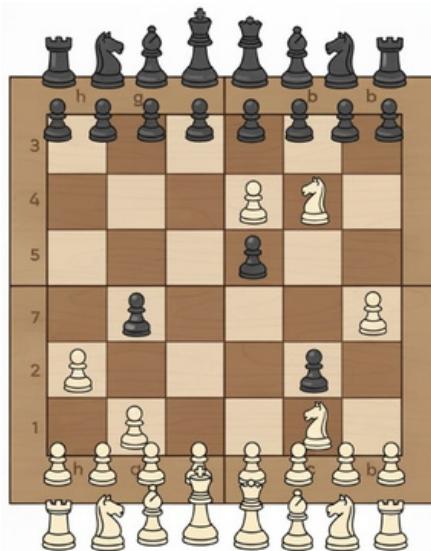
Event Sourcing, bir veri yapısı üzerinde gerçekleşen tüm değişikliklerin sistematik olarak kaydedilmesini öngören bir mimari desendir. Bu yaklaşım, ilgili verinin yalnızca güncel ve ham haliyle saklanması yerine, söz konusu verinin yaşam döngüsü boyunca geçirdiği tüm transformasyonların da verinin ayrılmaz bir parçası olduğunu savunur ve bu nedenle bu değişimlerin de kalıcı olarak kayıt altına alınmasını gereklidir. Event istek sonucunda olayın-işin-işlemenin gerçekleşmesi durumudur. Örnek vermek gerekirse ; bir kullanıcının sistem üzerinden kayıt olması yazılımsal açıdan bir eventdir ve bu event 'Kullanıcı Oluşturuldu' şeklinde ifade edilir. Şimdi basit bir şekilde bir satranç tahtası üzerinden Event Sourcing yaklaşımını ele alalım :



Bu görselde, boş bir satranç tahtası ve tahtanın etrafında duran, oyuna başlamak üzere olan satranç taşları göreceğiz. Bu, sistemin "başlangıç durumunu" ve hiçbir olayın henüz gerçekleşmediğini temsil eder.



Bu görselde, yine aynı satranç tahtası üzerinde, birkaç hamle yapılmış ve bu hamlelerin oklarla veya basit numaralarla gösterildiği bir akış yer alıyor. Örneğin, Piyon E2'den E4'e, At B1'den C3'e gibi. Bu, sistemde gerçekleşen "olayları" (eventleri) ve bunların bir sırayla kaydedildiğini temsil eder. Mevcut tahta durumu, bu olayların bir birikimidir.



Son görselde, satranç tahtası üzerinde belirli bir noktada mevcut oyun durumu gösteriliyor. Etrafında ise, bu duruma ulaşmak için kaydedilen tüm olayların (hamlelerin) küçük ikonları veya listesi bulunuyor. Bu, mevcut "durumun" (state) aslında geçmişteki olaylar zincirinin yeniden oynatılmasıyla her zaman oluşturulabileceğini vurgular. Şimdi yukarıdaki akışı adım adım bir tablo halinde nasıl event sourcing metoduyla kaydedebileceğimizi incelersek ;

## Event Sourcing Tablosu: Satranç Maçı (Aggregate: Maç)

Yine event sourcing konusunu ele alırken karşımıza 2 farklı terim daha çıkacak , bunlar aggregate ve projection. Yukarıdaki tabloda da dikkat ederseniz satranç oyununun tamamı bu hamlelerin bütününden meydana gelmektedir ve bir hamle veya bir kaç hamle satranç oyununun tamamını temsil etmemektedir. İşte buradaki durum yazılımsal süreçlerde bir veri için de meydana gelmektedir , veri doğası gereği zaman içerisinde değişiklik göstermeye meyillidir diyebiliriz.

**Aggregate :** Event Sourcing yaklaşımında tüm ilgili olayların ait olduğu ve sistemin tutarlılık sınırlını belirleyen bir birimdir. Bir veritabanında güncel durumu saklamak yerine, bir Aggregate (örneğimizde satranç **Maçı**), kendisine gelen bir komutu (örneğin bir hamle) işleyeceği zaman, kendi geçmiş olay akışındaki tüm olayları **baştan sona oynatarak** anlık durumunu yeniden inşa eder. Bu anlık durum üzerinden komutun geçerli olup olmadığını kontrol eder.

**Projection :** Aggregate'lerin oluşturduğu ham olay verisinden (Event Stream) türetilmiş, **sorgulama için optimize edilmiş** okuma modelidir. Olaylar, kronolojik ve yazma (write) için optimize edilmiş bir yapıdayken, son kullanıcılar için anlık ve hızlı sorgulanabilir bir durum (State) gereklidir. Projection'lar, olay deposundaki her yeni olayı dinler ve bu olayları kullanarak kendi okuma veritabanlarındaki tabloları/dokümanları günceller. Satranç örneğinde, **tahtanın o anki pozisyonu** bir Projection'dır; tüm geçmiş hamle olayları işlenerek tahtanın güncel durumunu gösteren, hızlı okunabilir bir görünüm sunar. Projection'lar sayesinde, sisteme sürekli olarak olayları yazılırken, okuma işlemleri hızlı ve performansı yüksek bir şekilde yapılabilir.

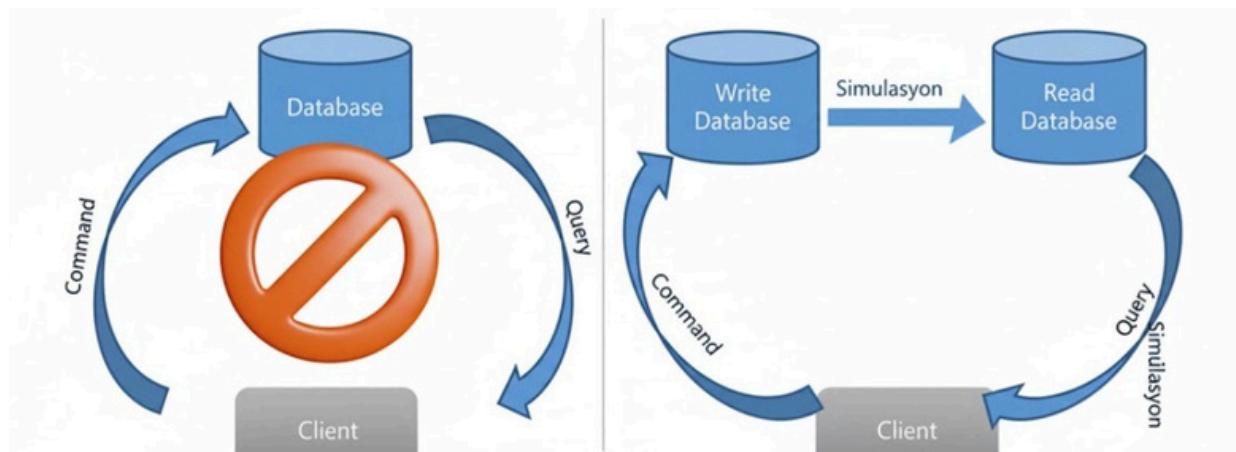
Event Sourcing yalnızca ileriye doğru yapılandırılmalıdır. Yanlış yazılan veya geriye alınması gereken durumlarda Update tarzı işlemler değil, yeni eventler oluşturulmalıdır.

# CQRS Pattern

CQRS (Command Query Responsibility Segregation), özellikle karmaşık ve büyük ölçekli yazılım sistemlerinde tercih edilen bir mimari yaklaşımıdır. Bu pattern'in temel prensibi, bir uygulamadaki **komut (command)** ve **sorgu (query)** işlemlerinin birbirinden ayrılarak, her birinin kendi model ve iş akışı üzerinden yönetilmesidir. Böylece sistem, hem performans hem de ölçeklenebilirlik açısından daha optimize bir yapı kazanır.

**Komut (Command):** Uygulamanın durumunda (state) değişiklik yaratan işlemleri temsil eder. Yeni bir kaydın eklenmesi, mevcut bir verinin güncellenmesi veya silinmesi gibi eylemler bu kapsamdadır. Komutlar genellikle "kesin" işlemleri ifade eder ve belirli, ölçülebilir sonuçlar üretir.

**Sorgu (Query):** Sistemde yalnızca veri okuma amacı taşıyan işlemleri ifade eder. Veritabanından bilgi çekmek, belirli kriterlere göre kayıtları listelemek veya tek bir kaydı detaylı biçimde görüntülemek gibi eylemler sorgu kapsamında değerlendirilir.



CQRS patterni sayesinde, komutlar (yazma işlemleri) ve sorgular (okuma işlemleri) birbirinden ayrılarak sistem yüküne göre bağımsız şekilde ölçeklendirme yapılabilir. Bu yaklaşım, özellikle yüksek trafiğe sahip uygulamalarda performans artışı ve kaynak kullanımında verimlilik sağlar. Örneğin, bir **e-ticaret sisteminde** ürünler kullanıcılar tarafından sürekli olarak görüntülenir ve listelenir; buna karşın yeni ürün ekleme veya mevcut ürünler güncelleme işlemleri çok daha seyrek gerçekleşir. Bu senaryoda, okuma ve yazma işlemlerinin tek bir veritabanı üzerinden yürütülmesi yerine, **okuma işlemleri için optimize edilmiş ayrı bir veritabanı** kullanmak, sistem performansını önemli ölçüde artırır. Böylece, sık gerçekleşen sorgu işlemleri yazma yükünden etkilenmez ve sistem genelinde daha dengeli, hızlı bir yapı elde edilir.

## Örnek Senaryolar :

E-Ticaret Uygulamaları : Bir e-ticaret sisteminde ürün kataloğunun görüntülenmesi, genellikle en yoğun gerçekleştirilen okuma işlemlerindendir. Bu katalog çoğu zaman sabit veya nadiren güncellenen bir veri yapısına sahiptir. Yönetici tarafından yeni ürün eklenmesi ya da kullanıcıların sipariş oluşturmalar gibi durumlar ise yazma (komut) işlemleri olarak öne çıkar. Bu noktada **CQRS deseni**, ürün kataloğuna yönelik okuma işlemlerini ayrı bir veri kaynağından yönlendirerek sistemi daha verimli ve ölçeklenebilir hâle getirir.

Sosyal Medya Platformları : Sosyal medya platformlarında kullanıcıların gönderi okuma işlemleri ile gönderi oluşturma, beğenme veya paylaşma gibi etkileşimleri arasında belirgin bir ayrımdır. Bu tür sistemlerde **CQRS deseni**, okuma ve yazma süreçlerinin birbirinden ayrılması için son derece uygundur. Yeni gönderi oluşturma veya bir gönderiyi beğenme işlemleri komut tarafına karşılık gelirken; gönderi akışının görüntülenmesi, kullanıcı profillerinin incelenmesi ya da etkileşim istatistiklerinin alınması sorgu tarafında ele alınabilir. Bu ayrımdır, sistemin hem ölçeklenebilirliğini hem de genel performansını artıran önemli bir mimari yaklaşımıdır.

**Bankacılık Uygulamaları:** Bankacılık sistemlerinde hesap bakiyesinin görüntülenmesi ile para transferi gibi işlemler arasında ciddi bir yoğunluk farkı bulunur. **CQRS deseni**, bu farklı yükleri birbirinden ayırarak sistemin performansını optimize etmek için etkili bir çözümüdür. Bu sayede, bakiye sorgulama işlemleri hızlı ve düşük gecikmeyle yanıt verebilirken; para transferleri veya hesap güncellemleri komut katmanında güvenli bir şekilde yürütülebilir. Sonuç olarak, sistem hem güvenilirliğini hem de verimliliğini artırmış olur. Bankacılık gibi yüksek güvenlik ve tutarlılık gerektiren alanlarda, bu ayırm özelliğe sistemin ölçeklenebilirliğini destekler. Ancak, **eventual consistency** davranışını dikkate alınmalı ve kritik senaryolarda uygun dengeleme stratejileri uygulanmalıdır.

## Distributed Sistemlerde Traceability

Günümüz yazılım yaklaşımlarında öne çıkan temel parametreler **performans**, **yüksek kullanılabilirlik**, **ölçeklenebilirlik** ve **güvenlik** olarak sıralanabilir. Bu parametreleri mevcut kaynaklarımız doğrultusunda en ideal hale getirebilmek için doğru bir **mimari tasarıma** ihtiyaç duyulmaktadır. Bu noktada, günümüzün tercih edilen yaklaşımı olarak **dağıtık sistemlerin (distributed systems)** inşa edilmesi gündeme gelmektedir. **Traceability (izlenebilirlik)**, bir dağıtık sistemdeki bileşenlerin gereksinimler doğrultusunda nasıl tasarlandığının, birbirleriyle olan ilişkilerinin ve sistem genelindeki işleyişin izlenebilmesini ifade eden bir kavramdır. Traceability sayesinde, geliştirilen bileşenlerin gereksinimlere ne kadar uygun şekilde inşa edildiği analiz edilebilir, sistemde yapılan değişikliklerin etkileri belirlenebilir ve hataların kaynağı daha hızlı tespit edilebilir. Bu sayede değişikliklerin nedenleri kolayca anlaşırlar, sistemin genel davranışları daha net bir şekilde yorumlanabilir. Kısacası, traceability; dağıtık sistemdeki işlemlerin ve verilerin yaşam döngüsü boyunca kayıt altına alınmasını, takip edilmesini ve doğrulanmasını sağlar. Bu süreçte her bir işlemin nerede başladığı, hangi bileşenler veya servisler üzerinden geçtiği ve nihai olarak hangi hedefe ulaştığı açıkça ortaya konur. Bu da sistemin şeffaflığını, yönetilebilirliğini ve güvenilirliğini önemli ölçüde artırır.

### Traceability ve Log Farkı :

**Traceability** (izlenebilirlik), bir sistemdeki verilerin, işlemlerin veya değişikliklerin başlangıcından sonuna kadar hangi adımlardan geçtiğini, kim tarafından ve ne zaman gerçekleştirildiğini takip edebilme yeteneğidir; genellikle kalite kontrol, hata tespiti veya güvenlik açısından süreçlerin bütünlüğünü korumak için kullanılır. Log (kayıt) ise sistemde gerçekleşen olayların veya işlemlerin zaman damgasıyla birlikte kaydedildiği ham veridir. Kısacası, log izlenebilirliğin (traceability) temelini oluşturan veri kaynağıdır; traceability bu log'lardan anlamlı bir iz sürme ve ilişkilendirme yaparak sistemin geçmişini veya neden-sonuç zincirini analiz etme sürecidir.

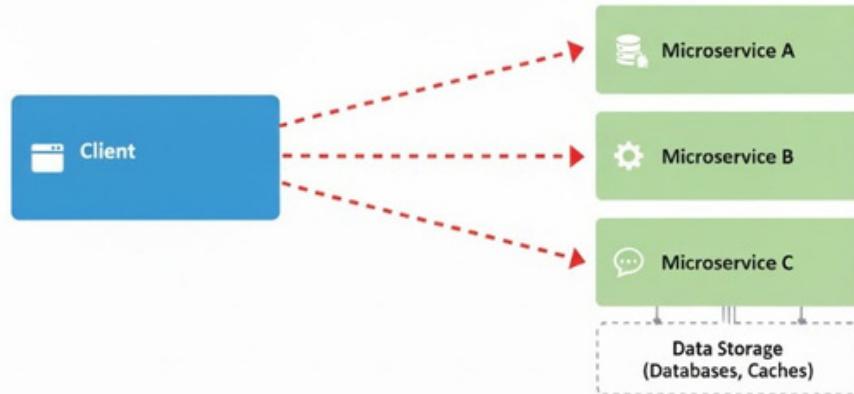
### Distributed Sistemlerdeki Traceability Bileşenleri :

Traceability sadece yazılımın kodsal parçası için tasarlanmış bir süreç değildir. Yazılım projesi seviyesinde daha geniş bir anlam ifade eden bir süreçtir. Dolayısıyla kodsal açıdan bir verinin izlenebilirliği, bileşenlerden yalnızca biridir. Proje sürecinde traceability açısından dikkate alınması gereken diğer boyutlardaki bileşenler de söz konusudur.

- **Gereksinim izlenebilirliği :** Sistem gereksinimlerinin izlenebilirliği, her bir gereksinimin sistemdeki bileşenlere nasıl dönüştüğünü göstermektedir. Böylece gereksinimlerin nasıl uygulandığı, değiştirildiği takip edilebilir.
- **Tasarım izlenebilirliği :** Tasarım izlenebilirliği ise ürünün tasarım sürecini ve hangi gerekliliğe dönüştüğünü göstermektedir. Böylece ürünün ürün gereksinimlerinin içinde nasıl etkilişime girdiğini, yeni akışım süreçlerinin belirleyip izleme bağlamına sahip rahat anlamanızı sağlamaktadır.
- **Kod izlenebilirliği :** Kod izlenebilirliği ile de tasarımın nasıl kodlandığını takip edebilir, değişikliklerin bileşenle nasıl eşleştirildiğini, değişiklik talebi ile yazılım bileşeninin nasıl yapıldığını takip edilebilir.
- **Test izlenebilirliği :** Bir bileşenin veya sistem özelliğinin nasıl test edildiğini öğrenmek için, Bir test senaryosunun nasıl uygulandığını, sonuçlarının nasıl analiz edildiğini ve hataların nasıl düzeltildiğini takip edebilirsiniz.

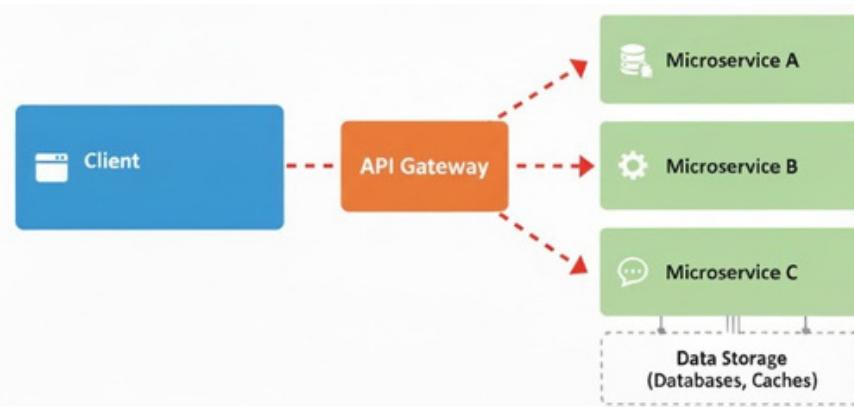
# API Gateway

API Gateway, istemciden(client) gelen ve farklı servislere yönlendirilecek istekleri tek bir giriş noktasında toplayan, ardından bu istekleri ilgili servislere ileten bir yapıdır. Kısacası, API Gateway; mobil uygulamalar, web tarayıcıları gibi istemcilerle arka planda mikroservisler arasında bir köprü görevi görür. API Gateway kavramını en iyi şekilde anlamadan yol, böyle bir yapı kullanılmadığında ortaya çıkabilecek sorunları incelemektir.



Göründüğü üzere, API Gateway kullanılmadığı durumlarda istemci ile servisler arasında — hatta servislerin kendi aralarında — doğrudan iletişim kurulması gereklidir. Bu da her bir istemcinin sistemdeki tüm servisler hakkında bilgi sahibi olma zorunluluğunu beraberinde getirir. Böyle bir yaklaşımın bir diğer zorluğu ise, her mikroservis için ayrı ayrı güvenlik politikalarının uygulanması gerekliliğidir. Kimlik doğrulama, yetkilendirme ve güvenlik kontrolleri gibi işlemler her servis için ayrı ayrı tasarlanmak zorunda kalır.

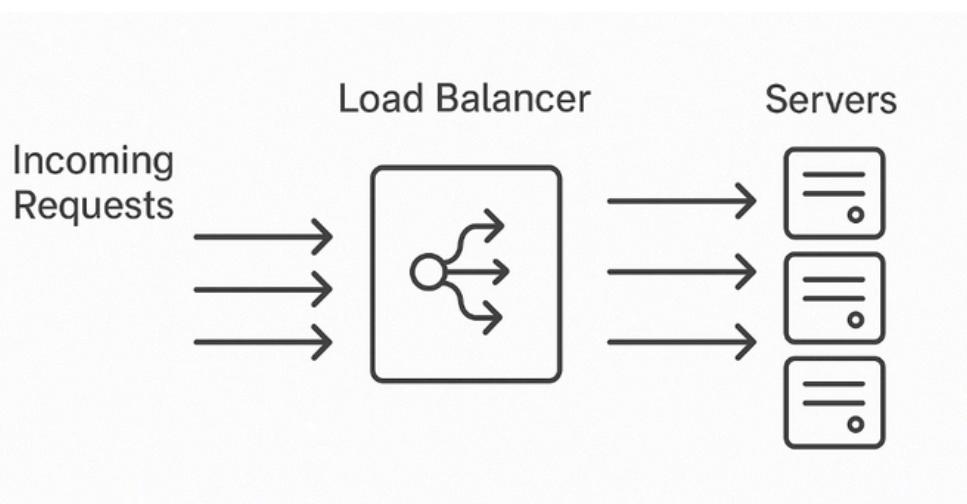
İşte API Gateway, bu tür problemleri çözmek için merkezi bir bileşen olarak görev yapar. Gelen ve giden tüm trafiği tek bir noktadan yönetir, güvenlik politikalarını burada uygular, protokol dönüşümlerini gerçekleştirir ve yük dengeleme(load balancing) işlevini yerine getirir.



API Gateway, görüldüğü üzere gelen tüm istekleri tek bir noktadan alır ve güvenlik, yetkilendirme ile kimlik doğrulama gibi kritik işlemleri bu katmanda uygular. Ayrıca, farklı protokoller arasında dönüşüm yapabilme yeteneğine sahiptir. Örneğin, gelen bir REST isteğini arka planda bir gRPC servisine dönüştürebilir. Bunun yanı sıra, API Gateway'in en önemli özelliklerinden biri de **API birleştirme (aggregation)** yeteneğidir. Bu sayede, bir istemcinin ihtiyaç duyduğu veriler birden fazla kaynaktan toplanarak tek bir yanıt hâlinde istemciye sunulur. Bu yaklaşım, istemcinin yaptığı istek sayısını azaltır ve sistemin genel verimliliğini artırır.

# Load Balancer

Load Balancing, bilgisayar ağları ve sistemlerinde gelen trafiği birden fazla sunucuya dağıtarak iş yükünü dengelemek için kullanılan bir tekniktir. Bu sayede tek bir sunucunun aşırı yüklenmesi önlenir ve kaynaklar daha verimli bir şekilde kullanılabilir. Aşağıdaki görselde de görülebileceği üzere, bir sunucu arızalandığında Load Balancer trafiği otomatik olarak diğer çalışan sunuculara yönlendirir. Böylece sistem kesintisiz bir şekilde çalışmaya devam eder. Ayrıca, trafik birden fazla sunucuya dağıtıldığı için her bir sunucunun üzerindeki yük azalır. Bu durum, hizmetin daha verimli ve yüksek performanslı bir şekilde sunulmasını sağlar. Ek olarak, bu yapı sayesinde sisteme kolayca yeni sunucular eklenebilir ve ölçeklenebilirlik (scalability) sağlanmış olur.



## Hangi Durumlarda Load Balancing Kullanılır :

- Yoğun trafik alan web siteleri ve uygulamalar: E-ticaret siteleri, sosyal medya platformları, haber siteleri gibi yüksek trafik alan sistemlerde, yüksek performans ve süreklilik gereksinimlerini karşılamak amacıyla load balancing tercih edilir.
- Dağıtık (distributed) sistemler: Birden fazla sunucu veya servisten oluşan dağıtık yapılarda, her bir bileşenin dengeli çalışması ve arızalara karşı dayanıklılığın artırılması için load balancing büyük önem taşır.
- Yedekli çalışma senaryoları: Verilerin ve hizmetlerin yedekli olarak çalıştığı sistemlerde load balancer, trafiği yedek sunuculara yönlendirerek olası bir arıza veya felaket anında hizmetin kesintisiz devam etmesini sağlar.

## Load Balancing Algoritmaları :

Load balancing algoritmaları, kullanılan teknolojiye ve mimariye göre davranışsal farklılıklar gösterebilir. Ancak genel olarak en yaygın kullanılan algoritmalar aşağıda özetlenmiştir.

### Round Robin :

Load balancing denince akla gelen ilk algoritmaların biridir. Gelen yükü tüm sunucular arasında sırayla dağıtır. Bu algoritmanın doğası gereği, her sunucuya eşit miktarda yük gönderilir. Dolayısıyla bu algoritmanın etkili bir şekilde çalışabilmesi için tüm sunucuların donanım gücü ve erişilebilirlik düzeylerinin birbirine yakın olması gereklidir.



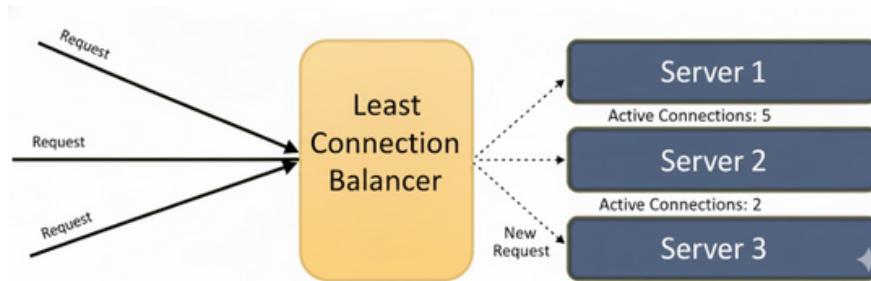
## Weighted Round Robin :

Round Robin algoritmasına kıyasla, farklı donanım gücüne veya erişilebilirlik düzeyine sahip sunucuların bulunduğu durumlarda daha dengeli bir yük dağıtımını sağlar. Bu algoritma, her sunucuya bir ağırlık (weight) değeri atayarak trafiği bu oranlara göre dağıtır. Böylece daha güclü veya yüksek kapasiteli sunucular, zayıf sunuculara kıyasla daha fazla istek alır.



## Least Connection :

Bu algoritma, gelen trafiği o anda en az aktif bağlantıya sahip sunucuya yönlendirir. Bu sayede özellikle trafik dalgalanmalarının sık yaşandığı sistemlerde yük dengesi daha etkin biçimde korunur ve sunucular arasında gerçek zamanlı denge sağlanır.



## IP Hash :

Bu yöntemde, aynı istemciden (IP adresinden) gelen istekler her zaman aynı sunucuya yönlendirilir. Bu yönlendirme, istemcinin ve sunucunun IP adreslerinden oluşturulan benzersiz bir hash değeri üzerinden yapılır. Böylece kullanıcı oturumlarının tutarlılığı korunur; örneğin, bir web uygulamasında kullanıcının sürekli aynı sunucuya iletişim kurması sağlanabilir.

