Microservices Temelleri

Monolitik mimarinin sınırları, mikroservis mimarisinin temel özellikleri, servis sınırlarını (bounded context) belirlemek, database per service, CAP teoremi ve Conway Yasası'na kısa bir bakış -- hiç kod yazmadan, kavramsal bir giriş.

Orta 20 dk

Son güncelleme: 6 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/microservices-fundamentals

---

Microservices Temelleri

Bu, `spring-boot` kursunun yeni "Mikroservisler" kategorisinin ilk dersi. Bu derste hiç kod yazmayacağız -- Spring Core ve Spring MVC kategorilerindeki derslerin aksine, burada amaç bir API'yi ya da bir mekanizmayı kod üzerinden göstermek değil, mikroservis mimarisinin *neden* var olduğunu, *hangi problemi* çözdüğünü ve *hangi yeni problemleri* beraberinde getirdiğini kavramsal olarak anlamak. Bu kategorinin bir sonraki dersinde ilk kez Spring Boot ile tek bir mikroservisi yapılandıracağız; ondan sonraki derste de iki servisin birbiriyle sade bir REST çağrısıyla nasıl konuştuğunu, gerçek ve çalıştırılabilir bir örnekle göreceğiz. Bu dersteki kavramlar, o iki dersin üzerine oturacağı temel.

Microservices Nedir?

Microservices (mikroservis mimarisi), bir uygulamayı tek, büyük ve birlikte deploy edilen bir birim yerine, her biri **kendi başına deploy edilebilen, kendi veritabanına sahip olabilen ve belirli, dar bir iş sorumluluğunu üstlenen** küçük, bağımsız servislerin bir araya gelmesiyle kurma yaklaşımıdır. Bu servisler birbirleriyle genellikle ağ üzerinden (HTTP /REST, mesaj kuyrukları gibi) konuşur -- aynı process içinde birbirini doğrudan metot çağrısıyla çağıran sınıflar değil, ayrı ayrı çalışan, ayrı ayrı deploy edilen bağımsız programlardır.

Örneğin bir e-ticaret sistemini düşün: sipariş yönetimi, envanter takibi, ödeme işleme ve kullanıcı hesapları -- bunların hepsi tek bir uygulama içinde, tek bir kod tabanında, tek bir deploy biriminde yaşayabilir (buna "monolit" diyoruz, bir sonraki bölümde detaylandıracağız) ya da her biri kendi kod tabanı, kendi deploy döngüsü ve kendi veritabanı olan ayrı birer servis olabilir (`order-service`, `inventory-service`, `payment-service`, `user-service`). Mikroservis mimarisi, ikinci yaklaşımı tanımlar.

Neden Var? (Monolitin Sınırları)

Mikroservisleri anlamanın en kolay yolu, onların çözmeye çalıştığı problemi anlamaktır -- ve bu problem küçük uygulamalarda neredeyse hiç yaşanmaz. Küçük ve orta ölçekli bir uygulama için monolitik mimari (tüm işlevin tek bir deploy biriminde toplandığı yaklaşım) genellikle **doğru** seçimdir: tek bir kod tabanı, tek bir build, tek bir deploy -- basit ve hızlı.

Sorun, uygulama ve onu geliştiren ekip **büyüdükçe** ortaya çıkar. Bir monolitte küçük bir değişiklik (örneğin sipariş onay mantığındaki bir düzeltme), teknik olarak yalnızca o modülü ilgilendirse bile, **tüm uygulamanın** yeniden build edilip yeniden deploy edilmesini gerektirir -- bu, o modülle hiç ilgisi olmayan bir hata riskini de beraberinde taşır. Onlarca geliştiricinin aynı kod tabanında aynı anda çalışması, merge çakışmalarını ve "kimin değişikliği neyi bozdu" belirsizliğini artırır. Ve bir bölümün (örneğin sipariş oluşturma) yoğun trafik alması, ölçeklendirmek için **tüm uygulamanın** yatay olarak çoğaltılmasını gerektirir -- oysa örneğin raporlama modülü hiç o kadar trafik almıyor olabilir.

Mikroservisler bu üç sorunu, uygulamayı bağımsız deploy edilebilen parçalara bölerek hedefler: bir servisteki değişiklik yalnızca o servisin yeniden deploy edilmesini gerektirir, farklı ekipler farklı servislerde birbirini engellemeden çalışabilir, ve yalnızca trafiği yüksek olan servis ölçeklendirilebilir. "Monolitik Mimarinin Özellikleri" ve "Monolith mi Mikroservis mi? Karar Kriterleri" bölümlerinde bu değiş tokuşun (trade-off) hiçbir zaman bedelsiz olmadığını, mikroservislerin de kendi yeni problemlerini getirdiğini göreceğiz.

Tarihçe

"Microservices" terimi, birbirinden bağımsız, küçük servisler fikrinin kendisinden daha yeni. Fikrin kökleri 2000'lerin ortasındaki **SOA**'ya (Service-Oriented Architecture, Servis Odaklı Mimari) uzanır -- SOA da uygulamayı servislere bölmeyi savunuyordu, ama genellikle ağır, merkezi bir "Enterprise Service Bus" (ESB) üzerinden haberleşen, karmaşık kurumsal standartlarla (SOAP, WS-* protokolleri) çalışan bir yaklaşımdı.

"Microservices" terimi, bugünkü anlamıyla ilk kez 2011 civarında bir yazılım mimarları atölyesinde telaffuz edildi, ama asıl yaygınlaşması **2014**'te, Martin Fowler ve James Lewis'in birlikte yayınladığı "Microservices" başlıklı makaleyle oldu -- bu makale, SOA'nın ağır merkezi altyapısından farklı olarak, **hafif** iletişim mekanizmalarını (genellikle düz HTTP /REST), her servisin kendi veritabanına sahip olmasını ve otomatik deploy süreçlerini (bugün "CI/CD" dediğimiz pratiklerin öncülü) vurguluyordu.

Aynı dönemde Netflix, Amazon ve benzeri büyük ölçekli teknoloji şirketlerinin, kendi devasa monolitlerini adım adım mikroservislere böldüklerini kamuya açık konuşmalarla paylaşması, fikri akademik bir tartışmadan somut, sektörde uygulanan bir pratiğe dönüştürdü. Bugün Spring Cloud (bu kategorinin ilerleyen derslerinde değineceğimiz Eureka, Spring Cloud Gateway gibi araçların ait olduğu ekosistem), Kubernetes gibi araçlar, o dönemde büyük şirketlerin kendi içlerinde elle çözdüğü birçok problemi (servis keşfi, yük dengeleme, deploy otomasyonu) standart, hazır çözümler hâline getirdi.

Monolitik Mimarinin Özellikleri

Devam etmeden önce "monolit" kelimesinin olumsuz bir sıfat olmadığını netleştirelim -- monolitik mimari, aşağıdaki özellikleriyle tanımlanan, kendi başına geçerli bir yaklaşımdır:

- **Tek kod tabanı:** Uygulamanın tüm modülleri (sipariş, envanter, ödeme, kullanıcı gibi) aynı repository'de, aynı proje yapısında yaşar -- tam olarak bu projenin (`learning-platform`) kendisinin bugüne kadar yapıldığı gibi.
- **Tek deploy birimi:** Uygulama derlenip tek bir çalıştırılabilir birim (bir `.jar`, bir container image) olarak paketlenir ve öyle deploy edilir.
- **Tek process, doğrudan metot çağrıları:** Modüller arası iletişim ağ üzerinden değil, aynı process içindeki doğrudan Java metot çağrılarıyla olur -- ağ gecikmesi yok, ağ hatası yok, dağıtık transaction yok (bu farkın önemini "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" bölümünde detaylandıracağız).
- **Genellikle tek, paylaşılan bir veritabanı:** Tüm modüller aynı veritabanı şemasını kullanır, tabloları doğrudan JOIN'leyebilir.

Küçük ve orta ölçekli birçok uygulama için bu, hâlâ **doğru** başlangıç noktasıdır -- basitliği, düşük operasyonel yükü (tek bir uygulamayı ayakta tutmak, onlarca servisi ayakta tutmaktan çok daha az iş gerektirir) ve hızlı geliştirme döngüsüyle. "Modüler Monolith": Bir Ara Yol bölümünde, monolitin bu basitliğini korurken iç organizasyonunu daha disiplinli hâle getiren bir yaklaşıma bakacağız.

Mikroservis Mimarisinin Temel Özellikleri

Mikroservis mimarisi, "Monolitik Mimarinin Özellikleri" bölümündeki dört özelliğin de tam tersini savunur:

- **Bağımsız deploy edilebilirlik:** Her servis, diğerlerinden habersiz, kendi başına build edilip deploy edilebilir. `order-service`'te yapılan bir değişiklik, `inventory-service`'in yeniden deploy edilmesini gerektirmez.
- **Dar, net bir sorumluluk:** İyi tasarlanmış bir mikroservis, tek bir iş yeteneğine (business capability) odaklanır -- "Servis Sınırlarını (Service Boundaries) Belirlemek" bölümünde bunu nasıl belirleyeceğimizi göreceğiz.
- **Kendi veri sahipliği:** Her servis, kendi verisinin tek sahibidir -- başka bir servisin veritabanına doğrudan erişmez (bkz. "Database per Service").
- **Bağımsız ölçeklenebilirlik:** Yalnızca trafiği yüksek olan servis, yalnızca kendisi yatay olarak çoğaltılabilir.
- **Teknoloji çeşitliliği (polyglot):** Farklı servisler, teorik olarak farklı diller/ framework'ler ile yazılabilir -- pratikte bu projede tüm servisler Spring Boot/Java olacak, ama bu, mimarinin izin verdiği bir esneklik.
- **Ekip özerkliği:** Her servisi ayrı bir ekip sahiplenebilir, kendi hızında geliştirip deploy edebilir -- Conway Yasası'nı işlerken ("Conway Yasası" bölümü) bu özerkliğin organizasyon yapısıyla nasıl iç içe geçtiğini göreceğiz.

Bu özellikler bedelsiz değil -- her biri, "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" bölümünde ele alacağımız yeni bir karmaşıklık kaynağı da getirir.

Bir Mikroservisin Anatomisi

Somutlaştıralım: `order-service` adında, siparişlerden sorumlu bir mikroservis düşün. Bu serviste tipik olarak şunlar bulunur:

- **Kendi API yüzeyi:** Dışarıya (diğer servislere veya bir ön yüze) açtığı REST uç noktaları -- örneğin `POST /orders`, `GET /orders/{id}`. Bu, servisin dışarıdan görünen, üzerinde anlaşılmış "sözleşmesi"dir.
- **Kendi iş mantığı:** Sipariş oluşturma, sipariş durumu güncelleme gibi, yalnızca siparişle ilgili kurallar -- envanter kontrolü ya da ödeme mantığı burada **değil**, ilgili servislerde yaşar.
- **Kendi veritabanı (ya da şeması):** Yalnızca `order-service`'in eriştiği, sipariş verisini tutan bir veritabanı.
- **Kendi çalışma zamanı:** Kendi process'i, kendi portu (örneğin `8081`), kendi `application.yml`'i -- bu kategorinin bir sonraki dersinde bunu gerçek bir Spring Boot projesinde kuracağız.

Aşağıdaki gibi, iki servisin yan yana nasıl konumlandığını gösteren basit bir taslak düşünebilirsin (bu bir kod değil, yalnızca kavramsal bir taslak):

```
order-service (port 8081)          inventory-service (port 8082)
├── OrderController                ├── InventoryController
├── OrderService (iş mantığı)      ├── InventoryService (iş mantığı)
└── kendi veritabanı (orders_db)   └── kendi veritabanı (inventory_db)
```

İki servis de bağımsız birer Spring Boot uygulaması -- biri çökse ya da yeniden deploy edilse, diğeri etkilenmeden çalışmaya devam eder (ama "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" bölümünde göreceğimiz gibi, birbirlerine ihtiyaç duydukları anlarda bu bağımsızlık kendi başına yeterli değildir).

Servis Sınırlarını (Service Boundaries) Belirlemek

Mikroservis mimarisine geçmenin en kritik ve en çok hata yapılan kararı, uygulamayı **nereden böleceğin**dir. Yanlış çizilen bir sınır, "Yaygın Hatalar" bölümünde göreceğimiz gibi, mikroservislerin tüm faydalarını götürüp yalnızca dağıtık sistemin bedelini bırakabilir.

Doğru yaklaşım, uygulamayı teknik katmanlara göre değil (örneğin "tüm controller'lar bir serviste, tüm repository'ler başka bir serviste" -- bu asla doğru değildir), **iş yeteneklerine (business capability)** göre bölmektir. "Sipariş yönetimi", "envanter takibi", "ödeme işleme" -- her biri, kendi başına anlamlı, iş dünyasında karşılığı olan bir sorumluluk alanıdır. İyi bir servis sınırı, şu soruya net bir cevap verir: "Bu servis, hangi iş kararını tek başına, başka bir servise sormadan verebilir?" `order-service`, bir siparişin oluşturulup

oluşturulamayacağına dair iş kurallarını (örneğin minimum tutar) tek başına bilir ve uygular -- ama envanterde ürün olup olmadığını bilmez, bunun için `inventory-service`'e sorar.

Bir sonraki bölümde ("Domain-Driven Design'a Kısa Bakış: Bounded Context"), bu fikri daha resmî bir çerçeveyle -- Domain-Driven Design'ın "bounded context" kavramıyla -- ilişkilendireceğiz.

> Sık verilen bir pratik tavsiye: eğer bir "servis"i tanımlarken sürekli "ve" bağlacı kullanman gerekiyorsa ("sipariş ve ödeme servisi" gibi), bu genellikle o servisin aslında iki ayrı sorumluluğu üstlendiğinin bir işaretidir.

Domain-Driven Design'a Kısa Bakış: Bounded Context

Domain-Driven Design (DDD), Eric Evans'ın 2003'te tanımladığı, karmaşık iş alanlarını yazılıma dönüştürmek için bir yaklaşım -- mikroservislerden çok önce ortaya çıktı, ama "Servis Sınırlarını (Service Boundaries) Belirlemek" bölümündeki soruyu yanıtlamak için bugün en sık başvurulan araçlardan biri hâline geldi.

DDD'nin bu bağlamda en önemli kavramı **bounded context** (sınırlı bağlam): büyük bir iş alanının (domain), kendi içinde tutarlı bir dil ve modelin geçerli olduğu, net sınırları olan bir alt parçası. Örneğin "ürün" kelimesi, envanter bağlamında "stokta kaç adet var, hangi depoda" anlamına gelirken, sipariş bağlamında "hangi fiyattan, kaç adet sipariş edildi" anlamına gelebilir -- aynı kelime, farklı bağlamlarda farklı anlamlar ve farklı veri modelleri taşıyabilir. İyi tasarlanmış bir mikroservis mimarisinde, her mikroservis genellikle bir (ya da yakından ilişkili birkaç) bounded context'e karşılık gelir.

DDD'nin bir diğer kavramı **aggregate** (bütünleşik küme), bir bounded context içinde birlikte tutarlı kalması gereken nesneler grubudur -- örneğin bir `Order` ve onun `OrderLine`'ları birlikte bir aggregate oluşturur, biri değişirken diğeri tutarsız bir durumda kalmamalıdır. Bu kursta DDD'yi derinlemesine işlemeyeceğiz -- yalnızca, "servis sınırlarını nasıl belirlerim?" sorusuna DDD dünyasının verdiği cevabın "bounded context'lere göre" olduğunu bilmen yeterli.

Database per Service

Mikroservis mimarisinin belki de en sık ihlal edilen kuralı budur: **her servis kendi verisinin tek sahibidir, başka bir servisin veritabanına doğrudan erişmez.** `order-service`, `inventory-service`'in veritabanındaki tabloları doğrudan sorgulayamaz -- envanter bilgisine ihtiyacı varsa, `inventory-service`'in API'sini çağırır (bunu "Servisler Arası İletişim" konusunda göreceğiz).

Bunun nedeni "Servis Sınırlarını (Service Boundaries) Belirlemek" bölümündeki fikirle doğrudan bağlantılı: eğer iki servis aynı veritabanı şemasını paylaşıyorsa, aralarındaki sınır yalnızca kod düzeyinde var demektir -- şemadaki bir değişiklik hâlâ her iki servisi de etkiler, tam olarak "Neden Var? (Monolitin Sınırları)" bölümünde kaçınmaya çalıştığımız sıkı

bağlılığın (tight coupling) aynısı, yalnızca iki ayrı process arasına taşınmış hâli. Bu duruma bazen **"distributed monolith"** (dağıtık monolit) denir -- mikroservislerin tüm operasyonel karmaşıklığını taşıyan ama monolitin hiçbir avantajını (basit, tek parça deploy) sağlamayan, en kötü iki dünya.

Database per service prensibi bunun bedelini de beraberinde getirir: artık `order-service` ile `inventory-service`'in verilerini tek bir SQL sorgusuyla JOIN'leyemezsin, ve iki servisin verisi arasında (eski, tek veritabanlı bir sistemdeki gibi) anlık, garanti bir tutarlılık yoktur -- bunun yerine **eventual consistency** (nihai tutarlılık) ile çalışılır. Bu ödünü "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" ve "CAP Teoremine Kısa Bir Bakış" bölümlerinde derinleştireceğiz.

> "Database per service" bir veritabanı sunucusu başına bir servis anlamına gelmek zorunda değil -- aynı PostgreSQL sunucusunda, her servisin kendi ayrı şemasına /veritabanına sahip olması da yeterlidir. Önemli olan fiziksel sunucu sayısı değil, **mantıksal izolasyon**: hiçbir servisin başka bir servisin tablolarına doğrudan erişmemesi.

Dağıtık Sistemlerin Getirdiği Yeni Zorluklar

Buraya kadarki bölümlerde mikroservislerin çözdüğü problemleri gördük -- şimdi madalyonun öbür yüzüne bakalım. "Monolitik Mimarinin Özellikleri" bölümünde saydığımız dört özellik (tek kod tabanı, tek deploy, doğrudan metot çağrıları, paylaşılan veritabanı) aynı zamanda monolitin **ücretsiz olarak** sağladığı garantilerdi -- mikroservislere geçince bunların hiçbiri bedava gelmez, hepsini kendin, ayrı ayrı çözmen gerekir:

- **Ağ güvenilmez olabilir:** Aynı process içindeki bir metot çağrısı neredeyse hiç başarısız olmaz; ama servisler arası bir HTTP çağrısı zaman aşımına uğrayabilir, ağ kesintisi yaşanabilir, karşı servis o an ayakta olmayabilir. Bu tür arızalara dayanıklı kod yazmak, kursun ilerleyen aşamalarında değinmeyi düşündüğümüz Resilience4j gibi kütüphanelerin var olma sebebi.
- **Partial failure (kısmi arıza):** Monolitte "uygulama ya tamamen çalışır ya da tamamen çöker" -- dağıtık bir sistemde ise bazı servisler çalışırken bazıları çökmüş olabilir. Sistemin geri kalanının bu durumda ne yapacağına (hata mi döner, eksik veriyle mi devam eder) baştan karar vermek gerekir.
- **Eventual consistency (nihai tutarlılık):** "Database per Service" bölümünde değindiğimiz gibi, servisler arası veri artık anlık olarak tutarlı değildir -- bir sipariş oluşturulduktan hemen sonraki milisaniyede, envanter servisi henüz güncellenmemiş olabilir. Sistemin bu kısa tutarsızlık penceresini nasıl yöneteceği, kursun ilerleyen olası bir konusu olan Distributed Transactions'ın (Saga pattern gibi yaklaşımlarla) odak noktası.
- **Gözlemlenebilirlik (observability) zorluğu:** Monolitte tek bir log dosyasına bakman yeterliyken, on servise yayılmış bir isteği takip etmek, merkezi loglama ve dağıtık izleme (distributed tracing) gerektirir.

- **Test etme zorluğu:** Tek bir servisi test etmek monolitteki bir modülü test etmekten farklı değildir, ama servisler arası entegrasyonu test etmek, diğer servislerin de ayakta olmasını (ya da sahte/mock versiyonlarının çalışmasını) gerektirir.

Bu zorlukların hiçbiri "mikroservis kullanma" anlamına gelmiyor -- yalnızca "mikroservis kullanınca yeni sorumlulukların da geldiğini bil" anlamına geliyor. "Monolith mi Mikroservis mi? Karar Kriterleri" bölümünde bu değiş tokuşu nasıl tartacağımıza bakacağız.

CAP Teoremine Kısa Bir Bakış

"Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" bölümünde bahsettiğimiz eventual consistency fikrinin arkasında, dağıtık sistemler teorisinden gelen bir sonuç var: **CAP teoremi**. Bu teorem, dağıtık bir sistemin aşağıdaki üç özellikten **aynı anda en fazla ikisini** garanti edebileceğini söyler (bir ağ bölünmesi -- partition -- yaşandığında):

- **Consistency (Tutarlılık):** Sistemdeki her okuma, en güncel yazmayı (ya da bir hatayı) görür -- hiçbir zaman eski/tutarsız veri dönmez.
- **Availability (Erişilebilirlik):** Her istek, başarılı olsun olmasın, mutlaka bir yanıt alır -- sistem asla yanıtsız kalmaz.
- **Partition Tolerance (Bölünme Toleransı):** Sistem, servisler arasındaki ağ bağlantısı kesilse bile (parçalar birbirini göremese bile) çalışmaya devam eder.

Gerçek dünyada ağ bölünmeleri (Partition Tolerance'ın ele aldığı durum) her zaman olabilir -- bu yüzden pratikte seçim, gerçekte Consistency ile Availability arasındadır: ağ bölündüğünde, sistem ya tutarlılıktan (bazı isteklere hata/bekleme döner, ta ki tutarlı olduğundan emin olana kadar) ya da erişilebilirlikten (isteğe hemen yanıt verir, ama verinin güncelliğinden tam emin olamayabilir) ödün verir. `order-service` ile `inventory-service` arasındaki iletişim kesildiğinde, sistemin ne yapacağına dair bu tür kararlar, mikroservis mimarisinin kaçınılmaz bir parçasıdır -- monolitte bu soru hiç sorulmaz, çünkü tek process içinde "ağ bölünmesi" diye bir şey yoktur.

Conway Yasası

1967'de bilgisayar bilimci Melvin Conway'in ortaya attığı bir gözlem, mikroservis mimarisi tartışmalarında sıkça anılır: **"Bir organizasyonun tasarladığı sistemler, o organizasyonun iletişim yapısının bir kopyasıdır."** Basitçe: yazılımın mimarisi, onu yazan ekiplerin nasıl organize olduğuna benzeme eğilimindedir.

Bunun mikroservisler için pratik sonucu şu: eğer bir şirkette "Sipariş Ekibi", "Envanter Ekibi" ve "Ödeme Ekibi" diye ayrı, birbirinden bağımsız çalışan ekipler varsa, bu ekiplerin ürettiği mimari doğal olarak `order-service`, `inventory-service`, `payment-service` gibi ayrı servislere doğru eğilim gösterir -- çünkü her ekip kendi servisini bağımsız deploy edebilmek ister. Tam tersi de doğrudur: tek, büyük ve sıkı koordinasyonla çalışan bir ekip, doğal olarak tek bir monolit üretmeye eğilimlidir, çünkü aralarında zaten sürekli senkron iletişim var.

Bazı organizasyonlar bu ilişkiyi tersine çevirip bilinçli olarak kullanır -- buna **"Inverse Conway Maneuver"** denir: istenen mimariye (örneğin bağımsız mikroservisler) ulaşmak için önce ekip yapısını o mimariyi destekleyecek şekilde (küçük, özerk, uçtan uca sorumlu ekipler) yeniden düzenlemek. "Mikroservis Mimarisinin Temel Özellikleri" bölümündeki "ekip özerkliği" maddesi, bu fikrin doğrudan bir uzantısı.

"Modüler Monolith": Bir Ara Yol

"Monolitik Mimarinin Özellikleri" bölümünde monolitin basitliğinin bir avantaj olduğunu, "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" bölümünde de mikroservislerin bedelsiz olmadığını gördük. Bu ikisi arasında, son yıllarda giderek daha çok önerilen bir ara yol var: **modüler monolith** (modüler monolit).

Fikir şu: uygulamayı hâlâ tek bir deploy birimi olarak, tek bir process'te çalıştır -- ama içeride, "Servis Sınırlarını (Service Boundaries) Belirlemek" bölümünde öğrendiğimiz aynı disiplinle, modülleri net sınırlarla, birbirinin iç detaylarına doğrudan erişmeden (yalnızca tanımlı arayüzler üzerinden) organize et. Her modül kendi paketinde yaşar, kendi verisine "sahip" olur (teknik olarak aynı veritabanında olsa bile, yalnızca kendi tabloları), ve modüller arası çağrılar bilinçli olarak dar bir arayüzden geçer.

Bu yaklaşımın cazibesi şu: mikroservislerin getirdiği ağ güvenilmezliği, partial failure ve eventual consistency sorunlarının hiçbirini yaşamazsın (hâlâ tek process, hâlâ doğrudan metot çağrıları) -- ama ileride gerçekten gerekirse (bir modülün trafiği/ekibi büyüdüğünde), o modülü ayrı bir servise **çıkarmak**, sınırlar zaten net olduğu için, baştan mikroservis olarak başlamaktan çok daha kolay olur. Birçok deneyimli mimar, "önce modüler monolith, gerektiğinde mikroservise böl" sırasını, "baştan mikroservislerle başla" yaklaşımına tercih eder -- bir sonraki bölümde bu tercihi nasıl değerlendireceğimize bakacağız.

Monolith mi Mikroservis mi? Karar Kriterleri

Buraya kadarki bölümleri birleştirirsek, bu soru tek bir doğru cevabı olan bir soru değil -- bir değiş tokuş (trade-off) değerlendirmesi. Mikroservisler lehine ağırlık basan sinyaller:

- Uygulama ve onu geliştiren ekip gerçekten büyük (onlarca/yüzlerce geliştirici) ve "Neden Var? (Monolitin Sınırları)" bölümündeki deploy coupling/merge çakışması sorunları fiilen yaşanıyor.
- Farklı modüllerin trafik/ölçeklenme ihtiyaçları belirgin şekilde farklı (örneğin bir modül saniyede binlerce istek alırken diğeri günde birkaç yüz istek alıyor).
- Farklı ekipler, birbirlerini engellemeden, kendi hızlarında deploy edebilmek istiyor ("Conway Yasası" bölümü).
- Servis sınırları zaten net -- ya bir modüler monolitten çıkarılıyor (bkz. "Modüler Monolith": Bir Ara Yol bölümü) ya da domain zaten iyi anlaşılmış durumda.

Monolit (ya da modüler monolit) lehine ağırlık basan sinyaller:

- Küçük bir ekip (birkaç geliştirici) var -- "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" bölümündeki operasyonel yükü (on ayrı servisi izlemek, deploy etmek, hata ayıklamak) taşıyacak kapasite yok.
- Domain/iş kuralları henüz netleşmedi, servis sınırları nereden çizileceği belli değil -- "Servis Sınırlarını (Service Boundaries) Belirlemek" bölümündeki gibi net bir cevap yoksa, yanlış yerden bölünmüş bir mikroservis mimarisi, "Yaygın Hatalar" bölümünde göreceğimiz gibi geri dönüşü zor bir hataya dönüşebilir.
- Uygulama henüz küçük/orta ölçekli, gerçek bir ölçeklenme ya da ekip-koordinasyonu problemi yaşanmıyor.

Pratikte deneyimli birçok mimar, "en baştan mikroservislerle başlamayı" değil, "'Modüler Monolith': Bir Ara Yol" bölümündeki yaklaşımla başlayıp, gerçek bir ihtiyaç (ölçüde, ekipte ya da deploy sıklığında somut bir sıkıntı) ortaya çıktığında kademeli olarak mikroservislere geçmeyi önerir.

Best Practices

- **Servis sınırlarını iş yeteneklerine (business capability) göre çiz, teknik katmanlara göre değil** -- bkz. "Servis Sınırlarını (Service Boundaries) Belirlemek". "Tüm controller'lar bir serviste" gibi bir bölünme, mikroservislerin hiçbir faydasını sağlamaz.
- **Her servisin kendi verisinin tek sahibi olmasını garanti et** -- başka bir servisin veritabanına asla doğrudan erişme (bkz. "Database per Service"); ihtiyacın olan veriyi her zaman o servisin API'si üzerinden iste.
- **Mikroservislere "çünkü moda" diye geçme -- gerçek bir sinyali bekle.** "Monolith mi Mikroservis mi? Karar Kriterleri" bölümündeki sinyallerin (ekip büyüklüğü, farklı ölçeklenme ihtiyaçları, deploy çakışmaları) fiilen yaşanmasını bekle.
- **Emin değilsen, modüler monolitle başla.** "Modüler Monolith": Bir Ara Yol bölümünde gördüğümüz gibi, net sınırlarla organize edilmiş bir monolitten mikroservise geçmek, baştan yanlış çizilmiş mikroservis sınırlarını düzeltmekten çok daha kolaydır.
- **Dağıtık sistemin getirdiği yeni sorumlulukları (ağ güvenilmezliği, eventual consistency, gözlemlenebilirlik) baştan kabul et** -- bunlar "sonra hallederiz" denip ertelenebilecek detaylar değil, mimarinin kendisinin bir parçası (bkz. "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar").
- **Ekip yapını, istediğin mimariyle uyumlu tut** -- "Conway Yasası" bölümünde gördüğümüz gibi, mimari ile organizasyon yapısı zaten birbirine benzeme eğiliminde; bunu görmezden gelmek yerine bilinçli kullan.

Yaygın Hatalar

1. "Distributed monolith" üretmek -- servisleri ayırıp veritabanını paylaşmaya devam etmek. "Database per Service" bölümünde gördüğümüz gibi, bu, mikroservislerin operasyonel bedelini (ağ, deploy karmaşıklığı) mikroservislerin hiçbir faydası olmadan üstlenmek demektir.

2. Servis sınırlarını teknik katmanlara göre çizmek (örneğin "tüm veri erişim kodu bir serviste, tüm iş mantığı başka bir serviste"). Bu, "Servis Sınırlarını (Service Boundaries) Belirlemek" bölümündeki iş yeteneği odaklı yaklaşımın tam tersi -- neredeyse her istek iki servis arasında gidip gelmek zorunda kalır.

3. Küçük bir ekiple, henüz gerçek bir ihtiyaç yokken baştan onlarca mikroservisle başlamak. "Monolith mi Mikroservis mi? Karar Kriterleri" bölümündeki sinyaller (gerçek ekip /ölçek/deploy baskısı) yokken bu, yalnızca "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" bölümündeki bedeli, hiçbir faydası olmadan ödemek demektir.

4. Dağıtık sistemin getirdiği yeni zorlukları (ağ hataları, kısmi arıza, eventual consistency) yok saymak, monolitteki gibi her şeyin her zaman anında tutarlı ve kullanılabilir olacağını varsaymak. "CAP Teoremine Kısa Bir Bakış" ve "Dağıtık Sistemlerin Getirdiği Yeni Zorluklar" bölümlerinde gördüğümüz gibi, bu varsayımlar dağıtık bir sistemde geçerli değildir.

5. Servis sınırlarını, uygulama zaten büyüdükten sonra, tasarım yapmadan elle tahminle çizmek. "Domain-Driven Design'a Kısa Bakış: Bounded Context" bölümündeki gibi bir çerçeve kullanmadan çizilen sınırlar, genellikle zamanla sürekli değişen, birbirine sürekli müdahale eden servislere dönüşür.

Özet ve Terimler Sözlüğü

Mikroservis mimarisi, bir uygulamayı bağımsız deploy edilebilen, kendi verisine sahip, dar sorumluluklu servislere bölme yaklaşımı -- monolitin büyüdükçe yaşadığı deploy coupling, merge çakışması ve tekdüze ölçeklenme sorunlarını çözmeyi hedefler, ama karşılığında ağ güvenilmezliği, kısmi arıza ve eventual consistency gibi yeni sorumluluklar getirir. Öne çıkan noktalar:

- Monolit **kötü** değil -- küçük/orta ölçekli birçok uygulama için hâlâ doğru başlangıç noktası; sorun yalnızca uygulama ve ekip büyüdükçe ortaya çıkar
- Servis sınırları, teknik katmanlara değil **iş yeteneklerine (business capability)** göre çizilir -- Domain-Driven Design'ın **bounded context** kavramı bu konuda yaygın kullanılan bir çerçevedir
- **Database per service**: her servis kendi verisinin tek sahibi -- bunu ihlal etmek "distributed monolith"e yol açar
- Dağıtık sistemler; ağ güvenilmezliği, kısmi arıza (partial failure) ve **eventual consistency** gibi monolitte hiç var olmayan yeni problemler getirir -- bunun teorik temeli **CAP teoremi**
- **Conway Yasası**, mimarinin organizasyon yapısına benzeme eğiliminde olduğunu söyler
- **Modüler monolith**, monolitin basitliğini korurken net iç sınırlarla organize olmayı, gerektiğinde mikroservise bölmeyi kolaylaştıran bir ara yaklaşımdır

Karar kontrol listesi (bkz. "Monolith mi Mikroservis mi? Karar Kriterleri"):

- Gerçek bir ekip büyüklüğü/deploy çakışması sorunu var mı, yoksa varsayımsal mı?
- Modüllerin ölçeklenme ihtiyaçları gerçekten farklı mı?
- Servis sınırları net mi, yoksa domain hâlâ netleşmemiş mi?
- Dağıtık sistemin getirdiği yeni operasyonel yükü (izleme, deploy, hata ayıklama) karşılayacak kapasite var mı?

Terimler Sözlüğü

Microservices (Mikroservis Mimarisi) — Bir uygulamayı, her biri bağımsız deploy edilebilen, kendi verisine sahip, dar sorumluluklu servislerden oluşturma yaklaşımı.

Monolit (Monolith) — Uygulamanın tüm işlevinin tek bir kod tabanında, tek bir deploy biriminde toplandığı mimari.

Modüler Monolith — Tek bir deploy birimi olarak kalan, ama içeride net, disiplinli modül sınırlarıyla organize edilmiş bir monolit türü.

Business Capability (İş Yeteneği) — Bir organizasyonun yerine getirdiği, kendi başına anlamlı bir iş sorumluluğu (örn. sipariş yönetimi, envanter takibi); servis sınırlarını belirlemede kullanılan temel birim.

Bounded Context (Sınırlı Bağlam) — Domain-Driven Design'da, kendi içinde tutarlı bir dil ve modelin geçerli olduğu, net sınırları olan bir alt domain.

Database per Service — Her mikroservisin kendi verisinin tek sahibi olması, başka servislerin veritabanına doğrudan erişmemesi prensibi.

Distributed Monolith (Dağıtık Monolit) — Servislere bölünmüş ama aralarında hâlâ paylaşılan bir veritabanı ya da sıkı bağlılık olan, mikroservislerin bedelini taşıyan ama faydasını sağlamayan anti-pattern.

Eventual Consistency (Nihai Tutarlılık) — Dağıtık bir sistemde verinin anında değil, kısa bir süre sonra tutarlı hâle gelmesi.

CAP Teoremi — Dağıtık bir sistemin, bir ağ bölünmesi sırasında Consistency (Tutarlılık), Availability (Erişilebilirlik) ve Partition Tolerance (Bölünme Toleransı) özelliklerinden aynı anda en fazla ikisini garanti edebileceğini söyleyen teorem.

Conway Yasası — Bir organizasyonun tasarladığı sistemlerin, o organizasyonun iletişim yapısının bir kopyası olma eğiliminde olduğunu söyleyen gözlem.