

Mikroservis Mimari

Önsöz

Doğru aracı doğru zamanda ve doğru şekilde kullanmak, yazılım geliştirme sürecinin en önemli ve de en çok tecrübe gerektiren konularının başında gelir. Burada ki '**araç**' ile; bir tasarım kalıbı, bir programlama dili, bir veri tabanı, bir algorithmadan tutun, seçtiğiniz bir CI yazılımına kadar çok geniş bir yelpazeyi düşünmelisiniz. Bunların her birisine, sizi amacınıza ulaştıracak olan birer araç gözüyle bakmalısınız.

Mikroservis Mimari de hedefe giden yolda seçebileceğimiz araçlardan bir tanesi. Ancak bu seçimi yaparken bunun sadece teknik bir karardan ibaret olmadığını, aslında organizasyonel bir yönünün olduğunu da unutmamamız gerekiyor. Yani mevcut ekip veya ekiplerinizin bu mimariyi uygulamak için uygun bir yapıda olması gerekmekte. Uygun yapıda olmaktan kastımızı bir sonraki bölümde detaylandıracağız.

Mikroservis Mimari'yi, temel prensiplerine sadık kalarak en doğru şekilde uyguladığımızda getirdiği avantajlardan faydalanırken, yanlış ve yersiz kullanımı ise hem zaman hem de kaynak israfıyla sonuçlanacaktır. Bu yüzden işe koyulmadan önce tüm detaylarıyla tartışılması gereken, çok önemli bir karar olduğunu unutmamalıyız.

Bu e-book, kişisel bloğumda (<https://medium.com/@suadev>) yayınlamış olduğum Mikroservis Mimari konulu yazılarımin, ekleme ve çıkarmalar yapılmış versiyonlarından ve bazı yeni ek bölümlerden oluşmaktadır.

Amacım bu e-book'u olabildiğince güncel tutmak ve gelecek katkılarla hatalı veya eksik kısımları zaman içerisinde düzelterek faydalı bir Türkçe kaynak haline getirebilmek.

Yazım yanlışları veya teknik konulardaki geri bildirimler için şimdiden teşekkür ederim.

Faydalı olabilmesi dileğiyle.

Suat.

Yol Haritamız

Mikroservis Mimari 2011 yılındaki ilk telaffuzundan bugüne kadar popülarlığını gün geçtikçe artırdı. Birçok ekip mevcut **Monolith** uygulamalarının yenilenme sürecinde veya yeni başlanacak projelerde Mikroservis Mimari'yi tercih etmeye başladı. Bu bölümde, bu mimariyi uygulamadan önce bilmemiz gereken konular ve işe koyulduktan sonra bizleri nelerin beklediğine dair önemli noktalara değineceğiz.

Konuyu aşağıdaki 4 başlık altında inceleyeceğiz;

- Uygulamanın Mikroservis Mimari'ye Uygunluğu
- Mikroservis Mimari'ye Geçiş İçin Hazır mıyız?
- Mikroservis Mimari'yi Uygularken Yapılan Hatalar
- Tavsiyeler

Mikroservis Mimari'yi Uygulamalı mıyız?

Bu çok önemli soru üzerinde belki de günlerce düşünüp konuşmak gerekirken, buna lüzum dahi görmeden biraz da gaza gelerek, "**Mikroservis Mimari'ye geçiyoruz !**" diyerek, yeterli ar-ge yapmadan kolları sıvayan bir çok ekip olduğuna eminim. Aksi halde bu kadar "**başarısızlık hikayesi**" ortaya çıkmazdı diye düşünüyorum. Bu başarısızlık hikayelerinin anlatıldığı bazı makalelere, arama motorları üzerinden aratarak sizlerde kolayca ulaşabilirsiniz.

Bu bölümde madde madde "**Mikroservis Mimari'yi Uygulamalı mıyız?**" sorusuna nasıl cevap aramamız gerektiğine bakacağız. Eğer burada bahsedilen sorunlar veya iyileştirmeler sizin için geçerli değilse Mikroservis Mimari'yi uygulamak için geçerli bir sebebiniz yoktur diyebiliriz.

Belirli Bir Teknoloji / Dil Ekosisteminde Sıkışmak

İçerisinde çeşitli modüller barındıran **monolith** yapıda bir uygulama düşünelim. **Monolith** yapıdan dolayı tüm kodumuz tek bir proje yani tek bir **repository** içerisinde.

Talep doğrultusunda projenize yeni eklenecek olan bir özellik için görüntü işleme yapmanız gerekiyor. Kullanıcılarınızın ara yüzden yükleyeceği araba fotoğraflarından plaka tanımlaması yaparak sisteminizde bu plakanın kayıtlı olup olmadığına bakmanız gerekiyor. Bu durumda uygulamayı geliştirdiğiniz programlama dili ve teknolojilerle bu özelliği geliştirmeniz gerekmekte. Eğer diliniz bu iş için pek elverişli bir dil değilse, hem geliştirme eforunuz artacak hem de belki ciddi performans sorunları yaşayacaksınız.

Bu işlemi yapan bağımsız bir servis geliştirelim, ve bu servisi istediğimiz farklı bir dille (**go**, **python** vs..) geliştirerek hem performans kazanımı elde edelim hem de teknoloji **stack**'imizi genişletelim diye düşünmeye başladıysanız, Mikroservis Mimari sizin için uygun **olabilir**.

Yüksek Ölçeklenebilirlik

Yok biz böyle iyiyiz dediniz **monolith** yapıda devam ediyorsunuz ve kullanıcı sayınız da gün geçtikçe artıyor. Derken bizim görüntü işleme modülünün çok geç yanıt döndüğü şikayetleri gelmeye başlıyor. Mimariyi, gelen image'leri kuyruklayıp tek tek kuyruktan alarak işleme üzerine kurmuşsunuz. Önceleri performans sorunuz yokken sisteme bir **t** anında yüklenen fotoğraf sayısı arttıkça kuyrukta bekleme süreniz de uzadı ve neticede kullanıcılarınız mutsuz.

Bu durumda sunucunuzda kaynak arttırımı yapıp, uygulamanızı dikey ölçekleyebilir (**bir yere kadar**) veya bir kaç sunucu daha devreye alıp yatay ölçeklendirmeye gidebilirsiniz. Peki görüntü işleme modülü dışında kalan modüllerde de bu ölçeklenme ihtiyacı söz konusu mu? Cevabınız hayır ise, tek bir servisi dilediğinizce yatay/dikey ölçekleyebilme imkanını elde edeceğiniz Mikroservis Mimari'yi düşünebilirsiniz.

Kolay ve Hızlı Release Çıkabilme

"En ufak bir değişiklikte koca uygulamayı olduğu gibi deploy ediyoruz" tarzı cümleler kurmaya başladıysanız, veya geliştirme süreci tamamlanan ve acilen devreye alınması gereken bir özelliği, projedeki bağımlılıklardan ötürü yeterince hızlı bir şekilde devreye alamıyorsanız, sizin gözünüz gibi bakıp büyüttüğünüz monolith uygulamanızın irili ufaklı servislere bölünme zamanı gelmiş **olabilir**.

Mikroservis Mimari'ye Geçiş İçin Hazır mıyız?

Önceki bölümde bahsettiğimiz konulardan sonra bu mimariye ihtiyacınız olduğuna kanaat getirdiniz. Peki nereden başlamalısınız? Yine madde madde inceleyelim;

Güçlü Bir DevOps Ekibi

Mikroservis Mimari'yi uygulayabilmek için en önemli şeylerden birisi hatta belki de en önemlisi DevOps kültürünün benimsendiği ve hakkıyla uygulandığı bir ekip veya ekipler oluşturabilmektir.

Martin Fowler, Mikroservis Mimari'nin getirdiği operasyonel yükü taşıyabilmenin güçlü bir DevOps takımına sahip olmaktan geçtiğini söyler. Öncelikle, servislerinizin building, configuration ve deploying süreçleri için bir **CI/CD pipeline**'ı tasarlamanız ve hayata geçirmeniz gerekiyor. Aksi halde çok fazla operasyonel yükü karşı karşıya kalabilir, dolayısıyla çok fazla vakit (**nakit**) kaybına uğrayabilirsiniz.

İlk etapta bazı süreçleri manüel yürütebilirsiniz belki. Ancak servis sayınız arttıkça ve **production** ortamınızı hazırlama zamanı geldiğinde artık **full-automated** olmanız gerekecektir.

En Az Bir Cloud Provider Üzerinde Uzmanlaşmak

Servislerinizi **on-premise** olarak kendi organizasyonunuz içerisinde kendi alt yapı ve ekipmanlarınızla yayınlatabilirsiniz elbette. Her ne kadar tavsiye edilen yöntem bu olmasa da en azından ilk etapta bu şekilde ilerlenebilir.

Ancak ilerleyen zamanlarda ürününüz **live** olmadan önce bir **cloud provider** (aws, google cloud, azure vs.) ile, cloud-based mimariye geçerek, hem operasyonel yükünüzü azaltıp hem de çok trafiği olan servisleriniz için **auto-scale** tarifeyi uygulayabilirsiniz. Bizim sunucular ayakta mı, ram/cpu yetersiz mi kaldı gibi soruları hayatınızdan çıkarmak sizin elinizde.

En Az Bir Container Teknolojisi Üzerinde Uzmanlaşmak

Container ve **Container Orchestration** araçlarının kaynak kullanımı ve ölçeklenebilirlik konuları başta olmak üzere bizlere kazandırdıkları hepimizin malumu. Monolith yapıdaki bir uygulamada, yani tek bir uygulamamız varken bile fayda sağlarken, söz konusu Mikroservis Mimari olduğunda adeta vazgeçilmez bir hal alıyor demek herhalde abartı olmaz.

Yine ilk etapta bir Container teknolojisi olmaksızın da ilerlenebilir. Mikroservis Mimari'nin benimsenmesi ve prensiplerine uygun şekilde hayata geçirilmesinin ardından, sıra deployment mekanizmasını değiştirmeye geldiğinde sanal makinalardan container'lara geçiş süreci başlatılmalı ve ardından bir container orchestration aracıyla dönüşüm tamamlanmalıdır. Sanal makinalar yerine Container'ları kullanmanın avantajları bu yazının konusu olmadığından burada kesmekte fayda görüyorum.

Gelişmiş Monitoring ve Notification Araçları

Birbirinden bağımsız olarak hayatına devam eden ve uygulamanın büyüklüğüne göre onlarca hatta yüzlerce sayıda olabilen mikroservis'lerin anlık olarak monitor edilebilmesi son derece önemlidir. Bu servislerden herhangi birinde meydana gelecek bir sorunun en hızlı şekilde ilgili yerlere sms, e-mail vb. gibi kanallardan bildirimi, sorunun hızlıca çözümü ve uygulamanın hayatını sürdürebilmesi adına çok kritiktir. Dolayısıyla Mikroservis Mimari'yi uygulamadan önce bu ihtiyaca cevap verebilecek bir monitoring aracının da devreye alınması veya en azından ön araştırmasının yapılması şarttır.

Bu monitoring ve alert ihtiyacı için kullanılabilecek bir çok ücretli veya ücretsiz açık kaynak araç mevcuttur.

Her Mikroservis İçin İzole Bir Veri Tabanı

Mikroservis Mimari, her servisin kendine ait, diğer servisler tarafından doğrudan erişime kapalı, izole bir veri depolama alanı olmasını gerektirir.

Monolith bir uygulamanın servislere ayrıştırılması konusunda, veri tabanı kısmı ilk etapta pek düşünülme-yebiliyor. Servisler şekillenmeye ve monolith yapıdan kopmaya başladıkça veri tabanlarının izolasyonu konuşulmaya başlanıyor. Tam bu noktada mevcut veri tabanının büyüklüğü ve tasarım kompleksliği aşılması gereken büyük bir sorun olarak karşımıza çıkıyor. Eğer bu sorun bir şekilde aşılamazsa, ya Mikroservis Mimariden vazgeçiliyor (servisleri ayrıştırmak için boşa harcanan efor), ya da izole veri tabanı konusundan taviz verilerek her servis ortak bir veri tabanını kullanacak şekilde ilerleniyor, ki bu Mikroservis Mimari'nin temel prensiplerinden olan "bağımsızlık" prensibine aykırı bir durum. Bu yüzden Mikroservis Mimari'ye dönüşümde ilk önce veri tabanı kısmını tasarlamak ve mevcut veri tabanının servislere özel olarak ayrışıp ayrıştırılamayacağı, bu işin eforunun ne olacağı konusunda biraz kafa patlatmak gerekiyor.

Mikroservis Mimari'yi Uygularken Yapılan Hatalar

Bu mimariyi uygularken zaman içerisinde mimarinin temelini oluşturan prensipler ihlal edilebiliyor. Bunu iki nedene bağlayabiliriz. Birincisi temel prensipleri tam olarak anlayıp benimsemeden, aceleyle işe koyulmak. İkincisi, izole veri tabanı başlığı altında bahsettiğimiz istemeden de olsa bilinçli olarak verilen veya verilmek zorunda kalınan bazı tavizler. Şimdi bu mimariyi uygularken aklımızın bir köşesinde sürekli tutmamız gereken önemli noktalardan bahsedelim;

Servisler Arası Ortak Kütüphane (library) Kullanmamaya Çalışın

“E hani **DRY (don't repeat yourself)** prensibine ne oldu? Aynı kodları her serviste tekrar tekrar yazacak mıyız?” diye düşünebilirsiniz.

Öncelikle tekrarlı kod oranımızı mümkün olduğunca azaltmamız gerekiyor tabi, **DRY** prensibinde bahsedilen konu **business logic**'i içeren kod parçacığının sadece bir kere yazılması. Ancak Mikroservis Mimari'de işler biraz değişiyor. Eğer ortak kullanılan fonksiyonlarınızı bir kütüphane haline getirir ve servislerinizi bu kütüphanelere bağımlı hale getirirseniz mikroservice mimarinin sağladığı en büyük kazanımlardan birinden bir miktar taviz vermiş olacaksınız. Bu kazanım bir önceki bölümde prensip olarak dile getirdiğimiz, **bağımsızlık** prensibidir.

Unutmayın; bu mimaride her servis bağımsız olarak geliştirilip, yine bağımsız olarak **release** edilebilmelidir. Peki bu durumda servislerin ortak olarak kullandıkları kodlar için nasıl bir yol izlememiz gerekiyor? Şu 3 seçenekten birisini tercih edebiliriz;

1- **Servisler arası** tekrarlı kodu kabullen.(Bir noktaya kadar)

Duplication is better than the wrong abstraction.

10 Modern Software Over-Engineering Mistakes

2- Ortak kodlar **business logic** içeriyorsa yeni bir **shared service** oluşturun.

3- Eğer mümkünse, servisleri yeniden dizayn et ve yeni alt **Mikroservis**/ler oluşturun.

Bir Servisin Diğer Bir Servisin Verisine Erişim Yöntemi

İzole veri tabanı maddesinde biraz bahsetmiştik, burada biraz detaylandıralım. İzole'den kasıt, bir servisin veri tabanına erişimin sadece o servis üzerinden olmasıdır. Yani bir servis başka bir servisin veri tabanına doğrudan erişim sağlayamaz, erişmesi gerekiyorsa veri tabanının sahibi olan ilgili servis üzerinden erişmelidir. Yani bir client gibi http isteğinde bulunmalıdır. Bu kuralda yine ihlal edilebilen kurallar arasında sayılabilir.

Genelde performans endişesinden dolayı yapılan bu hatada, **A** servisi **B** servisine istekte bulunmak yerine, doğrudan B servisinin veri tabanına erişerek, Mikroservis Mimarinin **gevşek bağlılık (loosely coupled)** prensibinin ihlaline neden olmaktadır. Bu ayrıca **A** servisi daha kompleks ve bakımı zor bir hale getirecektir.

Authentication, Throttling Gibi İşlemlerin Merkezileştirilmemesi

Bu maddeyi Throttling (**Rate Limiting**) üzerinden ele alabiliriz. Çok sayıda Mikroservis 'e sahip bir sisteminiz var ve her uygulamada olduğu gibi sizin uygulamanız için de client istatistikleri çok önemli. Hangi client veya hangi kanallardan (web, mobil) hangi servisler ne sıklıkla tüketiliyor? Hangi servislere daha çok yük biniyor? gibi soruların cevaplarını merkezi bir yapı üzerinden almak en doğrusu. Bu yapılar **Api Gateway** olarak adlandırılan servislerdir. Bu yapılar ile client'lardan gelen tüm istekler bir veya daha fazla olabilen api gateway'ler üzerinden ilgili servislere erişir.

Data önce deneyimle fırsatı da bulduğum **IBM'in Datapower** isimli gateway'i ilk aklıma gelen örnek. Bunun yanında open source ve tamamen ücretsiz olan çok başarılı api gateway'ler de mevcut. (**Kong, Tyk** vs.)

Api gateway kullanılmadığı takdirde, Rate Limiting ve benzeri işlemler her servis özelinde tek tek geliştirilmek zorunda kalınacak ki bu da bir süre sonra servisleri daha kompleks bir hale getirerek bakım maliyetlerini artıracaktır. Bir diğer sorun tabi gereksiz harcanan efor olacaktır. Api gateway ile merkezileştirilebilecek işlemler, her yeni doğan Mikroservis için yeniden tek tek geliştirilmek zorunda kalınacaktır. Api gateway'ler oldukça yetenekli araçlardır, ve Mikroservis mimari'de kullanımlarının bir çok faydası vardır. Öyle ki tek başına ayrı bir yazı konusu olacak genişlikte olduğundan burada kesmekte fayda görüyorum.

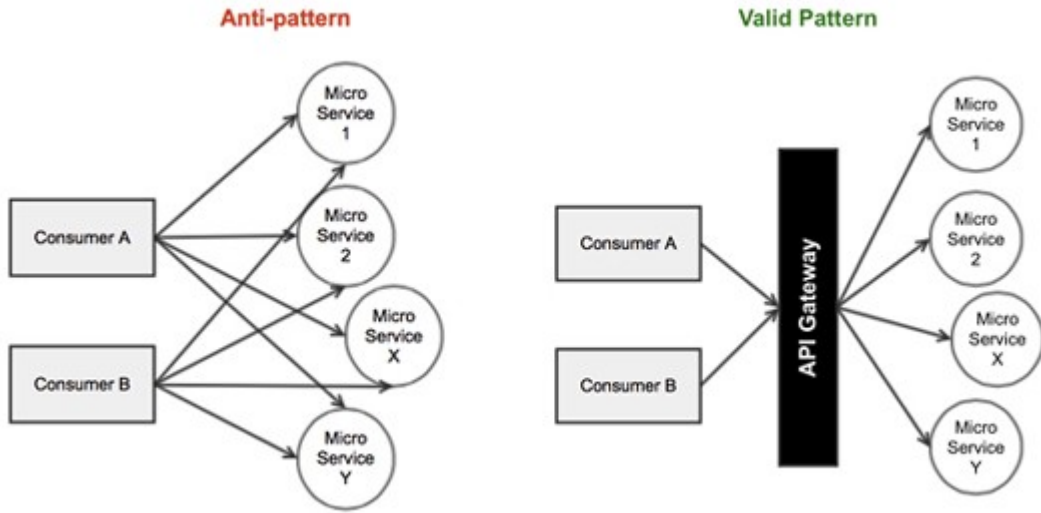


image from infoq.com

Tavsiyeler

Startup'lar veya olgunlaşmış ekiplerdeki yeni başlanacak olan projeler için tavsiyeler;

Eğer yazılım ekibiniz 2–5 kişilik bir ekipse ve ileride bu ekibi büyütme planlarınız yoksa, monolith başlamanızda fayda var. Zaten kısıtlı sayıdaki iş gücünüzü Mikroservis Mimari'nin getirdiği operasyonel yükün altında ezmek istemezsiniz.

Bu 2–5 kişilik ekibiniz için yakın zamanda büyüme planlarınız var, veya sıfırdan yeni ekip veya ekipler kuruyorsunuz diyelim. Bu durumda da aynı şekilde monolith olarak başlamanızı öneririm. Yani bana göre yeni uygulamalar için başlangıç her zaman monolith olmalı. Ekibiniz ve uygulamanız büyüdükçe, servislerinizin sınırları daha da netleşmeye başladıkça, DevOps alt yapınızı oluşturup, mimari değişikliğe gidebilirsiniz. Konuyla ilgili [buradaki](#) yazıya da göz atmanızı tavsiye ederim.

Monolith den Mikroservis Mimari'ye dönüşüm için tavsiyeler;

Bu işi kesinlikle hafife almayın ve sizi neyin beklediğini iyice araştırmadan ilk kazmayı vurmayın. Yukarıda mümkün olduğunca özet haline getirmeye çalıştığım maddelerden her birisi, işe başlamadan önce ve başladıktan sonra bir şekilde karşınıza çıkacaktır emin olun.

Mikroservis Mimari'ye dönüşümde Netflix'in hikayesi oldukça bilinen bir hikayedir. Monolith uygulamayı Cloud-Based **Mikroservis Mimari**'ye dönüştürme işine yanılmıyorsam 2009–2010 gibi başladılar ve bitirdiklerinde 2016 ya gelinmişti. Belki sizin sisteminiz bu denli büyük ve kompleks değildir, ancak işin ciddiyetini ve açılması gereken çok fazla kilit olduğunu Netflix'in hikayesinden anlamak mümkün.

Toparlarsak; Mikroservis Mimari'ye girmek aslında DevOps kafasına girmek olmalı diyebiliriz. Sadece teknik değil, organizasyonel ve kültürel bir değişim gerektirdiğinin bilincinde olunmalı. Geliştiricilerin sürecin her aşamasında etkin rol oynaması gerekiyor. Eğer daha önce hiç Mikroservis Mimari ve DevOps konuları hakkında deneyim edinmemiş bir ekip ile yola çıkıyorsanız, en azından temel konular hakkında bir eğitim ile işe başlamak bana göre elzemdir. Bu size vakit kazandıracak gibi daha emin adımlarla ilerlemenize de imkan sağlayacaktır.

Monolith'den Mikroservis'e

Bir önceki bölümde yeni bir projeye başlarken Monolith Mimari ile başlamanın daha doğru bir karar olacağını belirtmiş olsak da, Mikroservis Mimari ile başlangıç yapmak her koşulda yanlış olur gibi kesin bir ifade kullanmaktan da kaçınmalıyız. Dolayısıyla Mikroservis Mimari'ye geçiş için iki ana senaryomuz var diyebiliriz;

İlki, mevcut Monolith Mimarinin dönüşümü ve diğeri doğrudan Mikroservis Mimari ile başlama.

İlerleyen bölümlerde detaylıca inceleyeceğiz; Transaction Yönetimi, Servisler Arası İletişim, Veri Tabanı Tasarımı gibi konular her iki senaryo için de geçerli olmakla beraber, mevcut monolith mimarinin dönüştürülmesi senaryosu ekstra bazı zorluklar ihtiva etmekte. Bu bölümde biraz bunlardan bahsedeceğiz. **DDD ve Mikroservis Mimari** konu başlığında da bu dönüşüm konusuna atıfta bulunduğumuzu ayrıca belirtmek isterim.

Dönüştürme İşlemine Nereden Başlamalıyız?

Dönüşüm işleminin toplam süresi, dolayısıyla harcanacak toplam efor, uygulamanın bağımlılık ve karmaşıklık seviyesiyle doğrudan ilişkilidir. Birbirine sıkı sıkıya bağlı katmanlar ve modüllerden oluşmuş, ağırlıklı olarak iyi pratiklere uyulmadan geliştirilmiş bir uygulamanın dönüşüm sürecinin daha sancılı geçeceğini tahmin etmek zor değil. Bunun yanında modüler tasarlanmış ve gevşek bağlı, değişime kapalı ancak gelişime açık yapıdaki bir uygulamada ise işimiz daha kolay olacaktır. Buradan, modüler ve gevşek bağlı tasarımın ileride Mikroservis Mimari dönüşüm yapılması planlanan projeler için çok daha önemli hale geldiği sonucunu çıkarabiliriz.

Dönüşümü yapılacak olan uygulama hali hazırda kullanılmakta olan bir ürün ve mevcut monolith yapı üzerine aktif olarak geliştirmeler ve hata ayıklama işlemleri devam ediyor. Haliyle tüm bu geliştirme sürecini durdurup, Mikroservis Mimari dönüşümünü yapma gibi bir lüksümüz olmayacaktır. Nefes alan, üzerinde aktif olarak geliştirme yapılan bir uygulamayı dönüştürme gibi zorlu bir işimiz var.

Bu dönüşüm işini 'kodları birbirinden ayırma' gibi basit bir seviyeye indirgemek yapacağımız ilk yanlış olacaktır. Öyle ki, kodları servisler halinde ayırma işi bu dönüşüm sürecinin belki de en kolay işi olacaktır.

Öncelikle kısa ve uzun vadeli eylem planları hazırlamamız gerekmekte. Mevcut monolith uygulamadan kaç adet servis doğacak, ve bu servislerin sınırları neler olacak soruları cevap bekleyen en zor sorular olarak karşımıza çıkacaktır. Bu sorulara nasıl cevap vermeliyiz, servislerin sınırlarını nasıl çizmeliyiz gibi konulara **DDD ve Mikroservis Mimari** konu başlığında örnek bir senaryo üzerinden ele alacağımızdan, burada detayına girmiyoruz.

Uzun ve hararetli toplantılar sonunda servislerimiz ve bu servislerin sınırlarını belirledik. **Domain-Driven Design** ile ilerleyeceksek, Bounded-Context'lerimiz ve bunların sınırlarını doğru bir şekilde belirlemiş olmalıyız. Ne demiştik, mevcut monolith uygulama hayatına devam ediyor, öyleyse servislerimizi monolith yapıdan birer birer kopararak kontrollü bir şekilde devreye almamız gerekmekte. Dönüşümü tamamlayana dek, monolith bir uygulamamız ve belirli sayıda Mikroservis ile hayatımıza devam edeceğimiz anlamına geliyor bu.

Burada kişisel bir tavsiye olarak şunu söyleyebilirim; İlk servisi mümkün olduğunca basit ve ayırması kolay olan bir servis olarak seçmenizdir. Gerek servisi geliştirme süreci gerekse DevOps anlamında bu sizin acemilik döneminiz olacağından servisin basit olması hızlı çıktı almanızı sağlayacaktır. Genelde benim aklıma ilk gelen notifikasyon servisi oluyor. Hemen her uygulamada, e-posta ve sms gibi bildirimleri göndermekle yükümlü bir modülümüz oluyor ve bu modülün pek bir bağımlılığı da olmuyor diğer modüllerle. Dolayısıyla notifikasyon işini yapmakla sorumlu modülü monolith yapıdan koparıp istediğiniz bir teknolojiyle servis haline getirerek başlangıç yapabilirsiniz. Eğer bu servis bir veri tabanına ihtiyaç duyuyorsa, mevcut veri tabanınızdaki ilgili tabloları ayrı bir ilişkisel veya NoSQL veri tabanına taşıyarak tam izolasyonu sağlayabilirsiniz. Her servisin mutlaka kendine ait bir izole veri tabanı olması gerektiğini unutmamalısınız. Veri tabanını servisler özelinde parçalama konusuna Veri Tabanı Tasarımı konu başlığında yine örnek bir senaryo üzerinden daha detaylı değineceğiz.

Bir diğer önemli nokta ise, dönüşüm sürecindeyken gelecek yeni bir talebin konumlandırılması konusudur. Konumlandırmadan kastımız geliştirmenin nerede yapılacağı. Burada 3 farklı ihtimal söz konusu;

a) Mevcut Monolith **b)** Mevcut Mikroservis'lerden birisi **c)** Yeni bir Mikroservis

Eğer gelen bu yeni istek uygulamamız için tamamen yeni bir özellik anlamına geliyorsa yeni bir Mikroservis olarak geliştirmek önceliğiniz olmalı. Eğer mevcut modüllerden birisiyle bağlantılıysa doğru konumlandırmak o kadarda kolay olmayabilir. Eğer servis sınırlarımızı (Bounded Context'lerimizi de diyebiliriz) doğru bir şekilde çizdiyse, bu 3 seçenek arasında en doğru olanı çok zorlanmadan bulabiliriz. Aksi halde her yeni gelen istek için bu 3 seçenek arasında kalma ve yanlış kararlar verme ihtimalimiz olacaktır. Bu yüzden servis sınırlarının doğru olarak çizilmesi konusu büyük öneme sahip.

Transaction Yönetimi

Bu bölümün başlığı için birkaç farklı seçenek arasından bir seçim yapmak durumunda kaldığımı itiraf edeyim. **Transaction Yönetimi**, **Transaction Bütünlüğü**, **Veri Tutarlılığı** (Data Consistency) vb. gibi kavramların aslında aynı kapıya çıktığını söyleyerek başlayabiliriz.

Konuyu aşağıdaki başlıklara bölerek anlatmayı uygun buldum;

- Transaction ve Transaction bütünlüğü nedir?
- ACID prensipler hakkında
- Monolith uygulamalarda transaction yönetimi
- Mikroservis Mimari'de transaction yönetimi
- Mikroservis Mimari'de transaction yönetimi için **Two-Phase Commit** ve **Saga** tasarım kalıpları
- **Two-Phase Commit** vs. **Saga**

Transaction Kavramı

Transaction kelime anlamı olarak iş/işlem anlamına gelmekle birlikte kullanıldığı alana göre farklı anlamlar kazanabilmekte. Bankacılık sektöründe, yapılan bir EFT için kullanılırken, muhasebe dünyasında deftere yapılan her bir yazma işlemi için kullanılabilir. Veri tabanı üzerinde yapılan işlemlerin her birisi bizim için bir **transaction**'dır.

Bazı business transaction'lar, birden fazla transactionın çalışmasını gerektirebilir. Eğer Mikroservis Mimari söz konusuysa, bu aslında birden fazla servisin ard arda çalışması anlamına gelir. Bu arda arda çalışan transaction'lar dizisinin yönetilmeye ihtiyacı vardır. Yönetiminden kastımızın ne olduğuna bir örnek senaryo üzerinden bakalım.

Bir e-ticaret sitesinde bir ürünün siparişinden müşteriye teslim edilmesine kadar geçen sürede bir çok sürecin dolayısıyla transaction'ın işletildiğini biliyoruz.

Örneğin ödeme işlemi ve sonrasında ürünün stoktan düşülmesi süreçlerini ele alalım. Ödeme işlemi başarılı olmadan, stoktan düşme süreci ve sonraki süreçler işletilemez. Peki ödeme işlemi başarılı olduktan sonraki süreçlerin birisinde bir hata meydana gelirse ne

yapmalıyız? Yazılım tarafında bu durumu nasıl yöneteceğiz? Bu hata oluşuktan sonra o ana kadar veri tabanı üzerinde yapılmış olan işlemlerin tümünü geri almak gibi bir sorunumuz var. İşte bu sorun ve çözümü **transaction bütünlüğü/tutarlılığı/yönetimi** konusunun temelini oluşturmakta.

ACID Prensipler

ACID, değişikliklerin bir veri tabanına nasıl uygulanacağını yöneten 4 adet prensip sunar. Bunlar, **Atomicity**, **Consistency**, **Isolation** ve **Durability** prensipleridir. Bir kaç cümle ile açıklamak gerekirse;

Atomicity: En kısa ifadesiyle ya hep, ya hiç. Arda arda çalışan transaction'lar için iki olası senaryo vardır. Ya tüm transaction'lar başarılı olmalı ya da bir tanesi bile başarısız olursa tümünün iptal edilmesi durumudur.

Consistency: Veri tabanındaki datalarımızın tutarlı olması gerekir. Eğer bir transaction geçersiz bir veri üreterek sonuçlanmışsa, veri tabanı veriyi en son güncel olan haline geri alır. Yani bir transaction, veri tabanını ancak bir geçerli durumdan bir diğer geçerli duruma güncelleyebilir.

Isolation: Transaction'ların güvenli ve bağımsız bir şekilde işletilmesi prensibidir. Bu prensip sıralamayla ilgilenmez. Bir transaction, henüz tamamlanmamış bir başka transaction'ın verisini okuyamaz.

Durability: Commit edilerek tamamlanmış transaction'ların verisinin kararlı, dayanıklı ve sürekliliği garanti edilmiş bir ortamda (sabit disk gibi) saklanmasıdır. Donanım arızası gibi beklenmedik durumlarda transaction log ve alınan backup'lar da prensibe bağlılık adına önem arz etmektedir.

Monolith Uygulamalarda Transaction Yönetimi

Monolith mimaride transaction yönetimi Mikroservis Mimariye kıyasla oldukça kolaydır. Bir çok framework veya dil transaction yönetimi için kendi içlerinde bazı api'lar içerirler. (dotnet

için **TransactionScope** class'ı gibi) Bu api'lar tüm uygulamanın tek bir veri tabanına sahip olduğu, dolayısıyla tüm transaction'ların tek bir context üzerinde çalıştığı senaryolar için geliştirilmişlerdir. Yani monolith mimarilerde bu api'lar ile basitçe **commit** ve **rollback** işlemlerini yapabiliyoruz.

Commit işlemi **scope**'a dahil edilen tüm transaction'lar başarıyla çalıştığında en son yapacağımız işlem iken, **rollback** ise scope'dak herhangi bir transaction'da bir hata oluşması durumunda tüm işlemi iptal etmek için kullanılır.

Transaction scope içerisinde işletilen transactionlar **commit** edilene kadar diske yazılmadan memory'de tutulurlar ve eğer herhangi bir **t** anında **Rollback** yapılırsa, scope içerisinde o ana kadar işletilmiş tüm transactionlar memory'den silinerek işlem iptal edilmiş olur. **Rollback** yapmadan, **Commit** edildiğinde ise diske (veri tabanına) yazılarak transaction başarıyla tamamlanmış olur.

Aşağıdaki ekran alıntısında **TransactionScope** kullanım örneğini görebilirsiniz. Burada söz konusu transaction 2 step'li bir transaction'dır. Örneğin ikinci transaction'da meydana gelecek bir hata **Complete** metodunun çalışmamasına, dolayısıyla birinci işlemin de diske yazılmamasıyla yani iptaliyle sonuçlanacaktır.

```
using (var scope = new TransactionScope())
{
    try
    {
        //Transaction 1: Insert into master table, get record's primary key

        //Transaction 2: Insert into child table with primary key

        scope.Complete();
    }
    catch (Exception ex)
    {
        scope.Dispose();
    }
}
```


Mikroservis Mimari'de Transaction Yönetimi

Mikroservis Mimari'ler dağıtık mimarilerdir ve dağıtık mimaride bir çok konu için tek ve kolay bir çözüm genelde yoktur. Kimlik doğrulamadan, loglamaya, caching'den integration test yazmaya kadar bir çok konu uzmanlık gerektiren zor konular olarak karşımıza çıkmakta. Transaction yönetimi konusunu da bu konulara dahil ettiğimizi söylememe gerek yok sanırım.

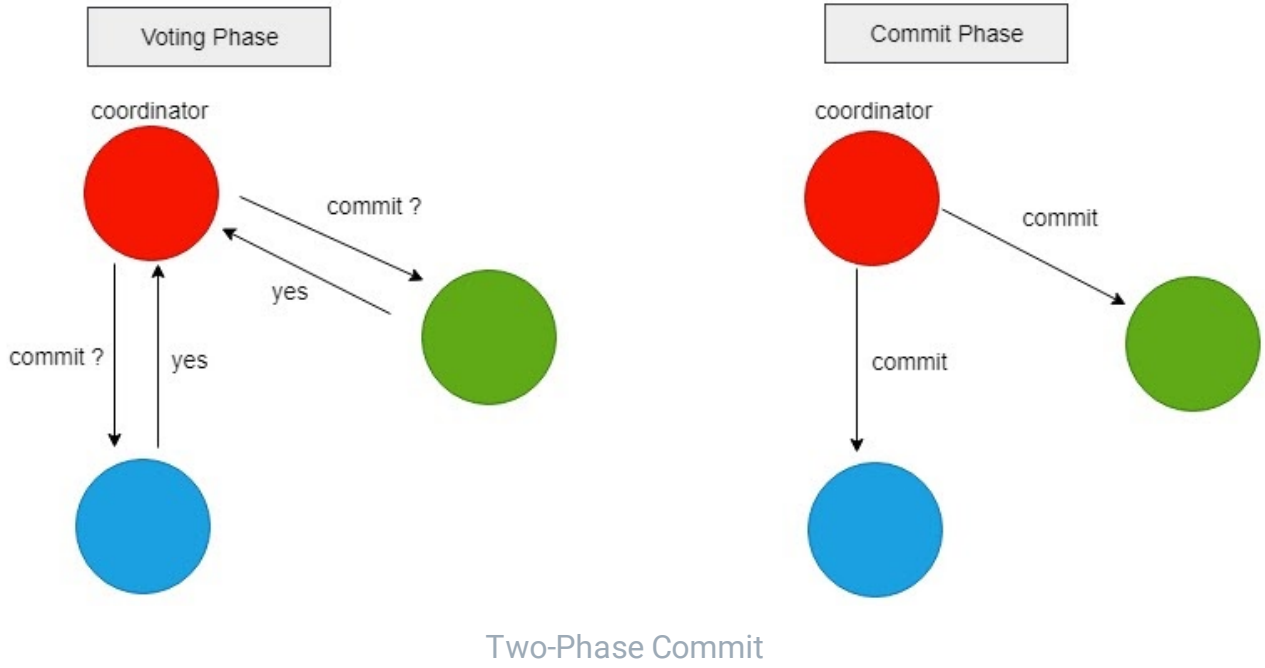
Mikroservis Mimari'lerde yukarıda bahsettiğim ACID prensiplerini korumak kolay bir iş değildir ve birden fazla yol mevcuttur diyebiliriz. Burada yalnızca **2PC** ve **Saga** konularına değineceğiz.

Two-Phase Commit (2PC)

2PC, dağıtık mimarilerde yukarıda bahsettiğimiz ACID prensipleri korumaya imkan sağlayan bir protokoldür. İsminden de anlaşılacağı üzere 2 fazdan oluşmaktadır ve bu 2 fazı yöneten bir **Kordinatör**'ümüz mevcuttur. İlk faz **prepare** (hazırlık veya oylama (voting)olarak da geçer), ikinci faz ise **commit** fazı olarak adlandırılır.

Yine e-ticaret örneği üzerinden açıklayalım. Ödeme ve stoktan düşme transaction'larını ele almıştık. Akış başladıktan ve transaction'lar tamamlandıktan sonra **Kordinatör**, hazırlık fazında bu 2 transaction'ın başarılı olup olmadığını yani **commit** işlemi için hazır olup olmadıklarını sorar. Eğer her iki transaction'dan commit için hazırız yanıtını alırsa 2. yani **commit** fazını icra ederek, işlemlerin kalıcı olarak diske yazılmasını sağlar.

Hata senaryosuna gelecek olursak; **Kordinatör**, birinci faz sonunda transaction'lardan **birisinden bile** commit edilemez yani hata oluştu bilgisini alırsa mevcut tüm transaction'ları iptal eder. Böylece işlem bir bütün olarak iptal edilmiş olur.



Saga Pattern

Dağıtık mimarilerde transaction yönetimi için kullanılan yöntemlerden birisi olan **Saga Pattern**, ilk olarak 1987 yılında [akademik bir makalede](#) ortaya atıldı.

Saga, her transaction'ın farklı ve bağımsız bir servis üzerinde lokal olarak çalıştığı ve yine o servis içerisinde verisini güncellediği transaction'lar dizisidir. Bu tasarım kalıbına göre, ilk transaction, dış bir etki ile (kullanıcının kaydet butonuna tıklaması gibi) tetiklenir ve artık sonraki tüm transaction'lar bir önceki transaction'ın başarılı olması durumunda tetiklenecektir. Transaction'lardan herhangi birisinde meydana gelecek bir hata durumunda ise tüm süreç iptal edilerek **Atomicity** presibine bağlılık sağlanmış olur. Biraz havada kalmış olabilir ancak aşağıdaki örnek senaryo çizimlerimle biraz daha net anlaşılacağını düşünüyorum.

Saga'yı uygulamak için bir kaç farklı yöntem mevcuttur. Ben **Events/Choreography** metodu ile Saga'yı servislerimiz arasında nasıl implemente edeceğimizden bahsedeceğim. Dilerseniz diğer bilindik yöntem olan **Command/Orchestration** metodunu da araştırabilirsiniz.

Events/Choreography Yöntemiyle Saga Uygulaması

Bu yöntemde ilk servis işini icra ettikten sonra bir **event** fırlatır ve bu event'ı dinleyen servis veya servisler tetiklenerek kendi local transaction'larını çalıştırır. Yani her servis aslında bir önceki servisin "ben işimi hallettim sıra sende" demesini bekler. Son servis çalıştıktan sonra artık bir event fırlatmaz ve süreç sonlanır.

Events/Choreography yöntemi, Saga'nın prensiplerine en uygun olan yöntemdir.

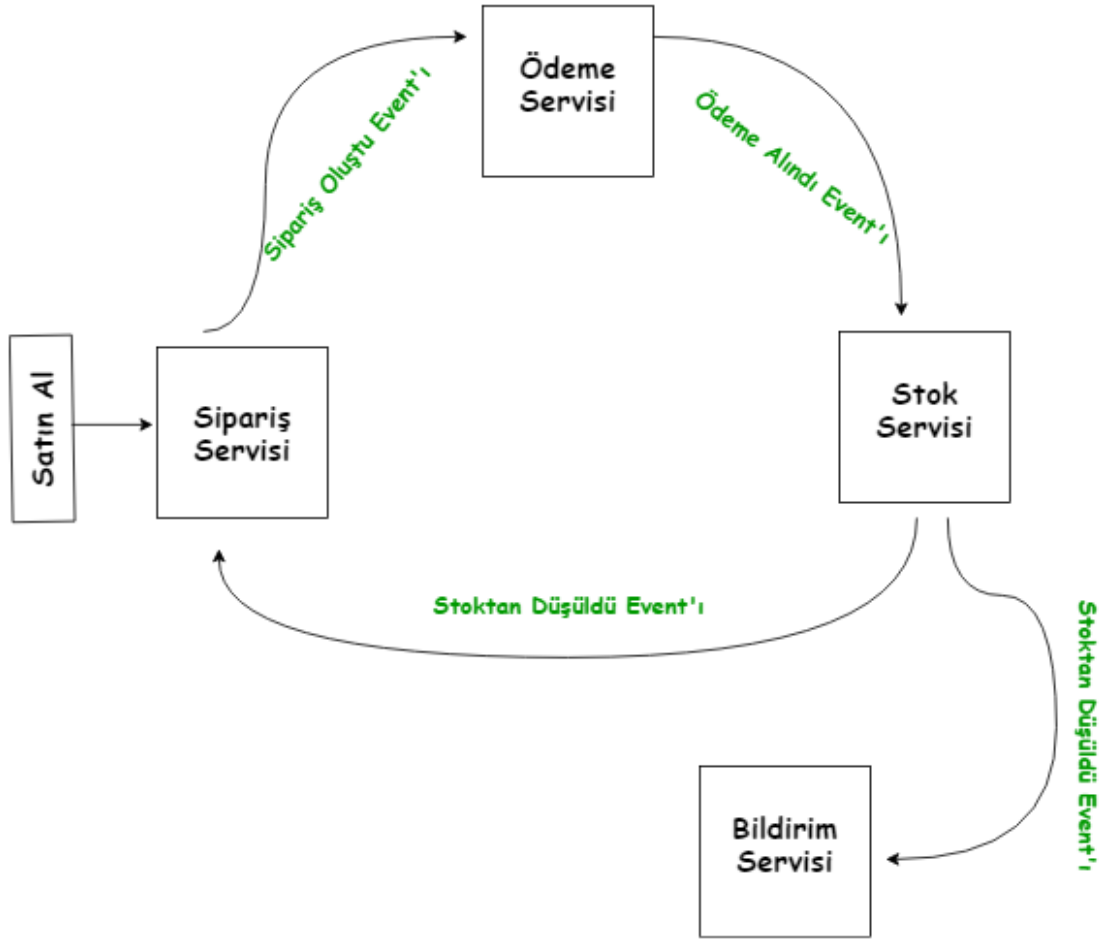
Uygulaması ve yönetmesi nispeten daha kolaydır. Transaction'lar birbirlerinden tamamen izoledir ve birbirleri hakkında bilgi sahibi olmak zorunda değildirler. Ancak bu yöntemde servislerimizin ve dolayısıyla event'lerimizin sayısı arttıkça sistemin karmaşıklığının da artacağını ve yönetilmesi zor bir hal alabileceğini unutmamak gerekiyor.

Yine sipariş verme örneğimiz üzerinden başarılı ve başarısız sipariş işlemlerini, iki farklı akış diyagramı üzerinden inceleyelim.

Örnek Başarılı İşlem Senaryosu (Commit)

Senaryomuz gayet basit. Hatırlarsanız ilk transaction'ımız harici bir müdahale ile tetikleniyordu. Kullanıcının ekrandan **Satın Al** butonuna tıklamasıyla;

- Saga'mızın ilk transaction'ı yani **Sipariş Servisi** tetiklenir.
- Sipariş servisi **Sipariş Oluşturuldu** Event'ini fırlatır.
- Bu event'i dinleyen **Ödeme Servisi** tetiklenir.
- Ödeme servisi **Ödeme Alındı** Event'i fırlatılır.
- Bu event'i dinleyen **Stok Servisi** tetiklenir.
- Stok servisi **Stoktan Düşüldü** Event'ini fırlatır.
- Bu event'i dinleyen **Bildirim ve Sipariş Servisleri** tetiklenir.
- Bildirim servisi kullanıcıya e-mail/sms gönderir.
- Sipariş Servisi siparişin durumunu **Başarıyla Tamamlandı** durumuna günceller.

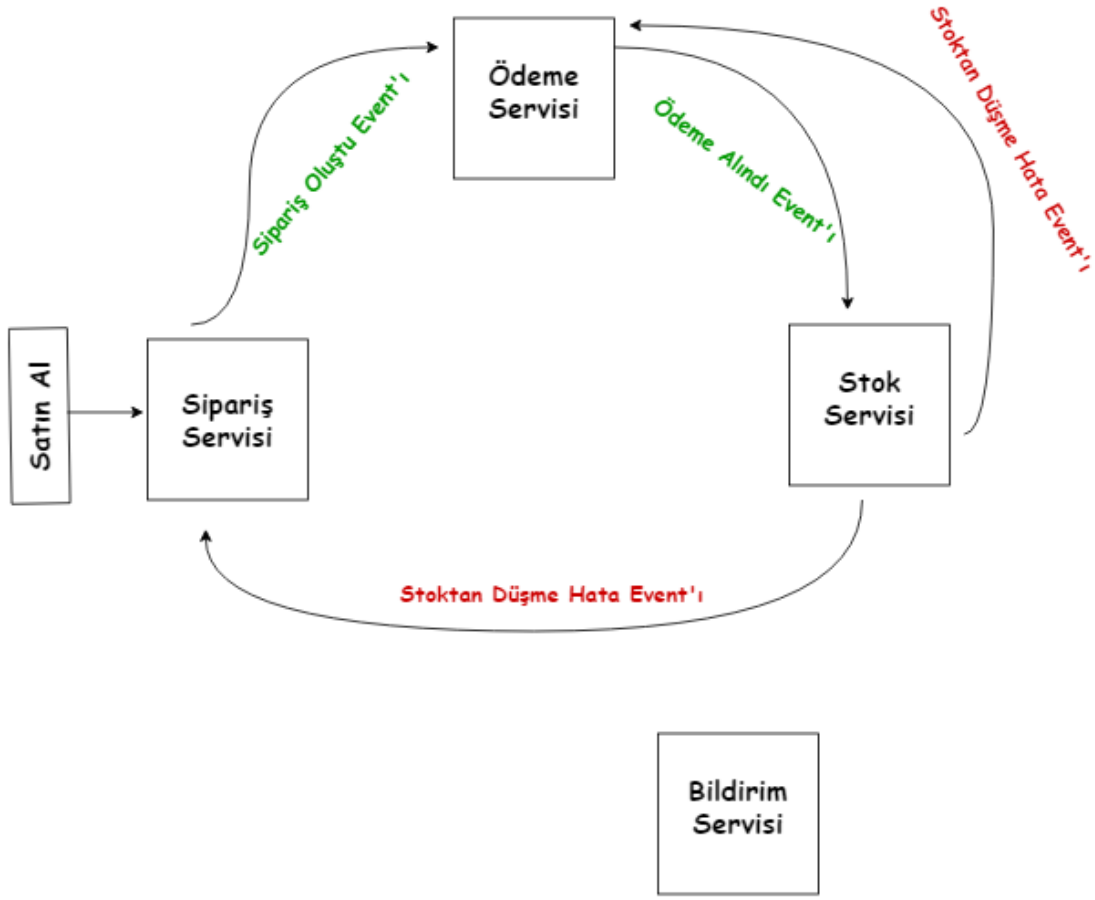


Başarılı sipariş Saga implementasyonu

Örnek Başarısız İşlem Senaryosu (Rollback)

Kullanıcının ekrandan **Satın Al** butonuna tıklamasıyla;

- Saga'mızın ilk transaction'ı yani **Sipariş Servisi** tetiklenir.
- Sipariş servisi **Sipariş Oluşturuldu** Event'ini fırlatır.
- Bu event'i dinleyen **Ödeme Servisi** tetiklenir.
- Ödeme servisi **Ödeme Alındı** Event'i fırlatılır.
- Bu event'i dinleyen **Stok Servisi** tetiklenir.
- Stok servisi **Ürün Stokta Yok Hata** Event'ini fırlatır.
- Bu event'i dinleyen **Ödeme ve Sipariş Servisleri** tetiklenir.
- Ödeme servisi kullanıcıya para iadesi yapar.
- Sipariş Servisi siparişin durumunu **Başarısız** durumuna günceller.



Başarısız sipariş Saga implementasyonu

Diyagramları Yorumlayalım

Öncelikle **Event Fırlatma** ifadesine aşina olmayanlarınız için biraz hava da kalmış olabileceğinden buna değinelim. Diyagramlarda belirttiğim Event'lerin her birisi aslında veri tabanına veya bir message queue'ya atılan bir kayıttan ibaret. Burada veri tabanı mı yoksa mesaj kuyruğu mu olacağı sizin tasarımınıza kalmış. Eğer ciddi yük altında çalışan bir uygulama söz konusuysa RabbitMQ, MsMQ veya Kafka gibi mesaj kuyruk yapıları kullanmanızı öneririm.

Gelelim **Event Dinleme** olayına. RabbitMQ Message Queue kullanılan bir mimaride açıklayacak olursak. Oluşan her event bir kuyruğa yazılır. Ödeme servisi **Sipariş Oluştur** event'ini dinliyor demek, bu event'in yazıldığı kuyruğa atılan her yeni mesajın Ödeme Servisi tarafından **Subscribe** edilmesi yani tüketilmesi demektir. Her servis için ayrı bir kuyruk olduğunu hayal edin. Hangi event'in hangi kuyruk veya kuyruklara yazılacağını

RabbitMQ'nun gelişmiş routing yapısıyla kolayca yönetebiliyorsunuz. Başarılı ve başarısız senaryo diagramlarını açıklayalım biraz daha net anlaşılacaktır.

Başarılı senaryo için çok fazla söylenecek bir şey yok aslında. Yalnızca **Stoktan Düşüldü** event'ine dikkatinizi çekmek isterim. Dikkat ederseniz bu event'i dinleyen iki servis var. Sipariş servisi ve bildirim servisimiz. Birkaç cümle önce '**kuyruk veya kuyruklara**' ifadesini kullanmıştık yani biz RabbitMQ'ya bir mesaj gönderirken bu mesajı birden fazla kuyruğa yaz diyebiliyoruz.

Gelelim başarısız senaryoya;

Stok servisine kadar her şey yolundaydı, ancak kullanıcının ekranında **Stokta Var** olarak gördüğü ürünün aslında stokta mevcut olmadığı ortaya çıktı ve neticede stok servisi **Stoktan Düşme Hata** event'ini fırlattı. Yani **hata durumunda bir önceki servis için event fırlatılır** diyebiliriz.

Tabi olay bu kadar da basit değil. Örneğin, ödeme servisinin hata alması durumunda sadece ilk servis için event oluşturulması yeterli olacakken, bildirim servisi için farklı bir tasarım yapmak gerekebilir. Neticede bildirim servisi **Saga** akışını bozan, işlemi sekteye uğratan bir servis değil. Yani oluşacak bir hatanın başka bir servis tarafından dinlenmesi gerekmeyebilir. Bildirim servisinde oluşacak hatalar için **retry policy**'ler tanımlayabiliriz.

Saga mı? 2PC mi?

Açıkçası ben her durumda Saga'yı uygulamayı tercih ederdim. Bunun en önemli nedeni, 2PC nin performans noktasında dezavantaj sağlaması. Hatırlayın, tüm katılımcılardan yanıt gelene kadar kaynakların lock'lı durumda bekletilmesi durumu. Ek olarak, 2PC ile gerçekleştireceğiniz her senaryoyu Saga ile de yapabilirken, tersi her zaman mümkün olmayabilir.

Saga'yı **Long Running Transaction** dediğimiz, transaction'ı birbirinden bağımsız ve asenkron çalışabilen step'lere bölerek yönetmenin doğru olduğu senaryolarda kullanırken, **2PC** nisbeten daha hızlı sonlanan ani transaction'lar için tercih edilebilir.

Saga'da bir sipariş işlemi step'lere (**sipariş-ödeme-stok-email vs..**) bölerek yönetmek kolaylık sağlıyor. Bunun yanında Saga'da her step'ten sonra commit işlemi yapıldığından, yani sonuç dış dünyaya gerçek olarak yansıdığından dolayı (**ödeme adımı müşteri**

ödeme alınması gibi) rollback işlemi daha kritik bir hal alıyor. Ödeme işleminin rollback yapılması yani müşteriye para iadesi yapılması esnasında oluşabilecek bir hata durumunda nasıl aksiyon almamız gerekir? Bu konu belki de Saga'nın en kritik konusu olabilir.

Veri Tabanı Tasarımı

Diğer bölümlerde olduğu gibi, bu bölümün de **monolith** mimari ile uygulama geliştirenler için de faydalı olacak bilgiler içerdiğini belirterek başlamak isterim.

Gerek Monolith'den Mikroservis'e geçiş sürecinde, gerekse sıfırdan Mikroservis Mimari ile geliştirmeye başlayacağınız projelerinize başlamadan önce üzerinde uzun uzun düşünüp, tartışıp, araştırmalar yapmanız ve cevap aramanız gereken en önemli 3 soru şunlar olacaktır;

- Kaç adet servisimiz olacak?
- 'Şu' modülü/özellği 'bu' servisten ayırıp, yeni bir servis mi oluşturmalı yoksa burada mı kalmalı?
- Hangi servis için hangi tip (RDBMS, NoSQL) veri tabanı seçmeli?

Dikkat ederseniz, yazının konusu veri tabanı seçimi olmasına rağmen servisleri nasıl ayırtmamız gerektiğinden bahsettik. Çünkü servisleri nasıl ayırttığınız, bir servisin tek bir iş mi (olması gereken), yoksa birden fazla iş mi yapması gibi konular, doğrudan seçeceğiniz veri tabanının tipine etki edecektir. İkinci bölümde örnek bir senaryo üzerinden incelediğimizde nasıl etki ettiğini daha net görmüş olacağız.

Konuyu üç ana bölümde ele alacağız;

- Veri Tabanı Tipleri ve Karşılaştırması
- Mikroservis Mimari'de Doğru Veri Tabanını Seçme
- Yanlış Seçim Yaptığımı Nasıl Anlarım?

Veri Tabanı Tipleri

Konu bir seçim yapmaksızın eğer, öncelikle alternatiflerimizin ne olduğunu bilmemiz gerekir. Bu alternatifler hakkında ne kadar derinlemesine bilgi sahibi olursak alacağımız kararlarda o kadar doğru olacaktır.

Bu bölümde ilişkisel veri tabanları ve yönetim sistemleri (RDBMS) ile ilişkisel olmayan (NoSQL-Not Only SQL) veri tabanlarından bahsedeceğiz. Her iki tip veri tabanından

bahsettikten sonra kısa bir karşılaştırma yapacağız.

İlişkisel Veri Tabanları ve Yönetim Sistemleri (RDBMS)

İlişkisel Veri Tabanları; veriyi diğer verilerle bir ilişki içerisinde tanımlayabilmemize ve erişebilmemize imkan sağlayan veri tabanlarıdır. Bir ilişkisel veri tabanında veri genellikle tablolar halinde tutulur. Tablolar satır ve sütunlardan oluşurlar. Bu tablo/satır/sütun yapısı, ilişkilerin kolay bir şekilde tanımlanabilmesine imkan sağlar.

İlişkisel veri tabanı yönetim sistemleri (**RDBMS**) ise ilişkisel bir veri tabanı oluşturma, güncelleme ve yönetme işlerini yapan yazılımlardır. Çoğu RDBMS veri tabanına erişim ve etkileşim için **SQL** (Structured Query Language) dilini kullanmaktadır.

İlişkisel veri tabanlarında bir tablodaki kayıtlar birbirleriyle ilişkili olduğu gibi tablolar arasında da ilişki olabilir. Bu ilişkiler RDBMS'lerin **veri tutarlılığını (data consistency)** sağlamasındaki en önemli etkidir. Tabi burada ilişkilerin sizin tarafınızdan doğru bir şekilde tanımlanmış olması önemlidir.

Veri Tutarlılığına Basit Bir Örnek

Musteri ve **MusteriDetay** isimli iki tablomuz olsun. Bu tablolar arasındaki ilişkiyi doğru olarak kurarsanız, **Musteri** tablosundan bir kayıt silerken size bu kaydın detay tablosundaki bir kayıtlı bağlantılı olduğunu ve o kayıt silinmeden silinemeyeceğini söyler. Alternatif olarak, bir müşteriyi sildiğinizde otomatik olarak detay bilgisi de silinmiş olur. Eğer bu ilişkiyi kurmamış olsaydınız, doğrudan **Musteri** tablosundaki bir kaydı silinebilir ve detay tablosundaki karşılığı silinmeden kalabilirdi. Bu kötü senaryonun gerçekleştiğini, yani **MusteriDetay** tablosunda var olan, ancak **Musteri** tablosunda hiçbir karşılığı olmayan detay satırlarının olduğunu düşünelim. İşte bu veri tutarlılığının bozulduğu duruma bir örnektir. Veri tutarlılığı, bu gibi durumların önüne geçilebilmesi için ortaya atılan bir kavramdır ve RDBMS'lerin vaad ettiği en önemli özelliklerindendir.

Bugün piyasaya baktığımızda en popüler ilişkisel veri tabanları olarak, MySQL (ücretsiz), PostgreSQL (ücretsiz), MSSQL, Oracle, IBM DB2 gibi veri tabanlarını sıralayabiliriz. Bu veri tabanlarından hepsi ilişkisel olmasına rağmen, bunlar arasında da yine seçim yaparken bilinçli bir seçim yapmak bizim faydamıza olacaktır.

Örneğin coğrafi bilgi sistemleri (GIS) ile alakalı bir proje geliştiriyorsanız, hiç düşünmeden PostgreSQL kullanmanızı öneririm. PostgreSQL içerdiği GIS spesifik veri tipleri ve eklentileri (<https://postgis.net>) ile bu alanda öne çıkmaktadır.

GIS örneğini verme sebebim; Yapacağımız seçimin bir RDBMS / NoSQL seçimden ibaret olmadığını, bunlardan birisinde karar kıldıktan sonra kendi içlerindeki alternatifler arasından da yine doğru tercihi yapmamız gerektiğini vurgulamaktı.

İlişkisel Olmayan Veri Tabanları (NoSQL)

İlişkisel veri tabanlarının geçmişi yaklaşık 40 yıl öncesine dayanır. Şüphesiz o zamandan bu zamana ilişkisel veri tabanları gelişim göstermektedir. Ancak yapı itibarıyla RDBMS'ler büyük boyutta verilerle başa çıkmak üzere ve bir **cluster** üzerinde dağıtık yapıda çalışmaya pek uygun değildir. Büyük miktarda veri ile başa çıkma konusunda cluster üzerinde daha efektif çalışan bir veri depolama teknolojisine olan ihtiyaç günden güne artarken, NoSQL kavramı bu ihtiyaca bir çözüm olarak 1998 yılında ortaya atıldı.

NoSQL camiası, ağırlıklı olarak, yüksek miktarda veri ve yüksek trafiğe odaklandı. **NoSQL, RDBMS'lerin sahip olduğu güçlü ve anlık veri tutarlılığından vereceği taviz ile daha yüksek performans ve erişilebilirlik elde etmeye yöneldi.**

Peki neydi bu **anlık/hızlı veri tutarlılığı**? Önceki bölümde verdiğimiz veri tutarlılığı örneği anlık veri tutarlılığına da bir örnek aynı zamanda. İlişkisel veri tabanlarında anlık veri tutarlılığı (**immediate consistency**) vardır. NoSQL de ise bu anlık, yerini nihai veri tutarlılığına (**eventual consistency**) bırakır. Bu konuyu son bölümdeki **Yüksek Seviye Veri Tutarlılığı** alt başlığında bir gerçek hayat örneği ile ele alacağımız için burada kesiyoruz.

CAP Teoremi

NoSQL'in performans ve erişilebilirlik için veri tutarlılığından taviz verdiğinden bahsettik. Taviz vermeden bahsetmişken, CAP teoremine de değinmek gerekiyor. Muhtemelen bir çoğunuzun daha önce bir şekilde denk geldiği düşündüğüm aşağıda görsel, teoremi tek başına açıklıyor aslında. Dikkat ederseniz ortada kırmızı bir çarpı işareti var, yani bu üç özelliğin tamamının bir NoSQL mimari de olamayacağını anlamamız gerekiyor. CAP'ın C,A,P sine birer cümle ile bakalım;

Consistency: Dağıtık sistemdeki tüm node'ların aynı veriye sahip olması durumu.

Availability: Sisteme yapılan her isteğin başarılı olsun başarısız olsun, bir yanıt alabilmesi durumu. (En güncel veriye sahip olmasa bile.)

Partition Tolerance: Mevcut node'lardan bir kısmında network veya başka bir sebepten ötürü bir sorun meydana gelerek erişilmez hale gelse bile, sistemin çalışmasına devam edebilmesidir.

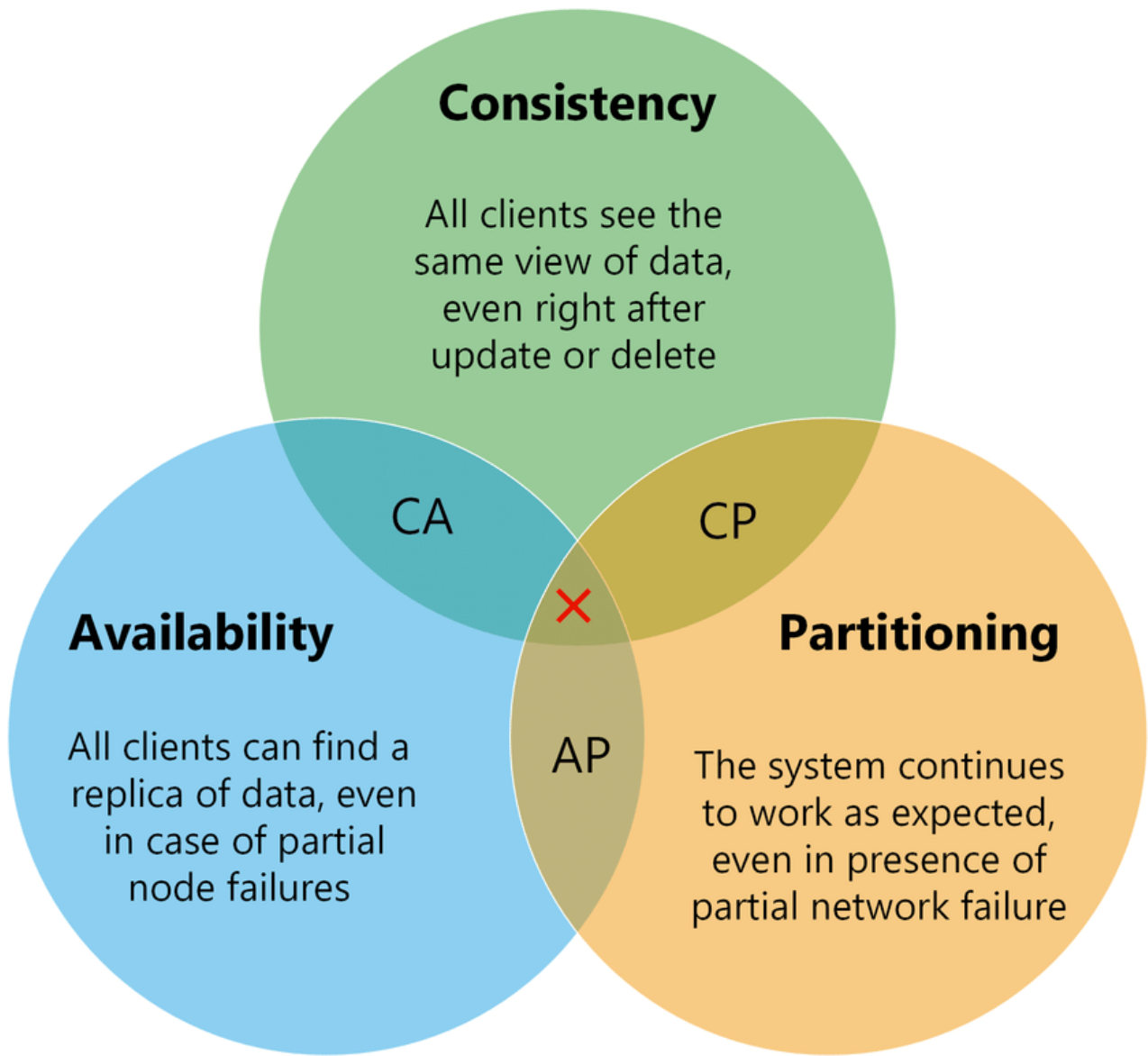


image from <https://www.researchgate.net>

Son olarak NoSQL veri tabanı çeşitlerine ve tercih edildikleri uygulama tiplerine bakacak olursak;

Document Based (MongoDB, CouchDB, etc.) E-ticaret siteleri, İçerik yönetim sistemleri vb.

Key/Value (Redis etc.) Kullanıcı oturum bilgisi saklama, Alış veriş sepeti verisi saklama vb.

Graph Based (Neo4J etc.) Sosyal medya uygulamaları, Graph tabanlı arama uygulamaları vb.

Column Based (Cassandra, HBase etc.) Transaction loglama, IoT uygulamaları vb.

RDBMS vs NoSQL

İki tür arasındaki farklılıklar çok ta karmaşık ve anlaşılması güç farklılıklar değil aslında. Bütün olay verinin nasıl saklandığı ve nasıl sorgulandığı meselesinden ibaret diyebiliriz. Şimdi alt başlıklar halinde karşılaştırmasını yapalım.

Ölçeklenebilirlik

RDBMS'lerinin yatay da ölçeklenebilmesi zor olduğundan, güçlü ve pahalı sunucularla dikeyde ölçeklendirme yoluna gidilir. NoSQL kolayca yatayda ölçeklenebileceğinden sunucu maliyetleri noktasında avantajlı **olabilir**.

Bu arada ölçeklenmenin zorluğundan bahsetmişken, geçenlerde Microsoft'un satın aldığı başarılı yerli girişim **Citus Data**'dan bahsetmek isterim. Citus Data aslında ilişkisel bir veri tabanı olan PostgreSQL'i dağıtık bir yapıda çalıştırmanıza imkan sağlıyor. Yani siz bir sorgu gönderiyorsunuz ve bu sorgu arka planda birden fazla node üzerinde paralel çalıştırılarak sonuç üretiliyor. Bu da yüksek performans demek tabi.

Açıkçası satın alma haberine kadar Citus Data'yı hiç duymamıştım. Bu teknoloji RDBMS'ler için devrim niteliğinde bir gelişme bana göre. Şöyle de bir blog post'a denk geldim, göz atmakta fayda var.

<https://www.citusdata.com/blog/2018/11/30/why-rdbms-is-the-future-of-distributed-databases/>

ACID Prensipler

NoSQL veri tabanlarının ACID olmadıkları yönünde aslında tam da doğru olmayan görüşler var. Önceki bölümde NoSQL veri tabanı tiplerinden bahsettik. Örneğin birçok Graph veri tabanı yapıları gereği ACID'dir. (Neo4J gibi.) Peki ya diğerleri? **CAP** teoreminden

bahsetmiştik. NoSQL veri tabanları çoğu zaman **AP** yi seçerek Strong Consistency'den taviz verebiliyorlar. Dikkat ederseniz strong ifadesini kullandım, yani eventual da olsa neticede bir Consistency sağlamış oluyorlar **AP**'yi seçtikleri zaman.

Dolayısıyla NoSQL veri tabanları ACID uyumlu değildir demek yanlış olacaktır. Kaldı ki ACID, Consistency'den ibaret değildir. ACID prensiplerden daha detaylı bahsettiğimiz transaction yönetimi bölümünü hatırlayınız.

Bakım Maliyetleri

RDBMS'lerin bakım maliyetleri yüksektir ve özellikle büyük ölçekli sistemlerde eğitilmiş insan gücüne olan ihtiyaç NoSQL'e göre daha fazladır. Bu da eğer danışmanlık alınıyorsa daha fazla danışmanlık maliyeti anlamına gelmektedir. NoSQL veri tabanları daha az yönetim ve onarım maliyeti getirir. Maliyet konusunda NoSQL veri tabanlarının bir çoğunun open source oluşu da önemli bir etkidir. Lisans ücretleri noktasında RDBMS ile ciddi fark vardır.

Olgunluk

RDBMS'ler çok daha eskiye dayandıkları için daha geniş bir topluluğa ve yetişmiş insan gücüne sahip olmakla birlikte oldukça **stabil** çalıştıklarını söylemek yanlış olmaz. Ek olarak hemen hemen tüm RDBMS'ler ortak bir dil olan **SQL** ile veri tanımlaması ve manipülasyonu yaptığından, bu dilde uzmanlaşan birisi için veri tabanları arasında geçiş yapmak kolaydır.

Big Data Uygulamalarında Kullanım

Direkt olarak şu tip veri tabanı büyük veri uygulamaları için daha uygundur demek yanlış olmakla birlikte, bazı projelerde her iki tür birlikte kullanılabilir. Veri, öncelikle performans ve ölçeklenebilirlik düşünülerek **unstructured** bir yapıda NoSQL veri tabanına kaydedilir ve asenkron olarak işlenerek **structured** bir yapı halinde ilişkisel veri tabanına yazılır. Ham verinin NoSQL de, işlenmiş olanın ise SQL de tutulması diyebiliriz yani. Böylelikle her iki veri tabanının da en önemli avantajlarından faydalanmış olunur.

Veri Tutarlılığı (Data Consistency)

RDBMS yüksek seviye veri tutarlılığı vaat ederken, NoSQL de durum böyle değildir. Önceki bölümde bir örnek vermiştik bu konuya ama bununla yetinmeyeceğiz ve bir sonraki

bölümde vereceğimiz sosyal medya uygulaması örneği ile daha net anlaşılacağından eminim.

Şema Bağımlılığı

NoSQL şema bağımsız olduğundan veri formatı uygulamaya çok fazla dokunmadan da değiştirilebilir. RDBMS'ler de **change management** büyük bir sorun haline gelebiliyor, özellikle kötü tasarlanmış ve ilişkileri doğru olarak kurulmamış veri tabanları için.

Sonuç olarak; RDBMS'ler NoSQL veri tabanlarından iyidir veya tam tersi doğrudur gibi söylemler kesinlikle yanlıştır. Her ikisinin de diğerine göre daha avantajlı olduğu senaryolar mevcuttur. Burada bize düşen hangi durumda hangisinin daha avantajlı olduğunu bilmek ve doğru seçimi yapabilmektir.

Mikroservis Mimari'de Doğru Veri Tabanını Seçme

Bildiğiniz üzere Mikroservis Mimari'yi uygulamak sizi bir programlama diline, framework'e, veri tabanına vs. bağlı kalmaktan kurtarıyor. Her bir servisinizi farklı farklı diller ve platformlarda bağımsız olarak geliştirerek yine bağımsız olarak canlıya alabiliyorsunuz. Tabi mimarinin temel prensiplerine sadık kalırsanız.

Bu dil, framework ve veri tabanı bağımsızlığı konusunda en çok göz ardı edilen konu veri tabanı konusu olabilir. Özellikle monolith'den Mikroservis'e geçişlerde, mevcut veri tabanı servisler özelinde parçalanırken, her Mikroservis için mevcut monolith veri tabanı ile aynı veri tabanını kullanmak ilk seçenek oluyor ve genelde de böyle ilerleniyor. En azından gözlemlerim ve okuduklarım kadarıyla durum böyle. Konfor alanımızdan çıkmak istemeyişimiz aklıma gelen ilk sebeplerden. Bunun bir neticesi olarak ilk hatayı yapmak kaçınılmaz oluyor. Şöyle ki;

Servislerimizi seçtiğimiz veri tabanının türüne göre ayırtırmaya başlıyoruz. Aslında yapmamız gereken bunun tam tersi olmalı. Servislerimizi veri tabanından bağımsız düşünerek, atomik ve en doğru şekilde tasarlayıp, veri tabanını ilgili servisin yapısına göre seçmeliyiz.

İyi güzel de hangi kriterleri baz alarak seçim yapmamız gerekiyor diye düşünebilirsiniz. Bizim için en önemli iki parametre, yüksek veri tutarlılığı ve yatayda ölçeklenebilme konularıdır.

Yüksek Seviye Veri Tutarlılığı ve Ölçeklenebilme Gereksinimi

Servisiniz için yüksek seviye **Consistency** önemliyse RDBMS kaçınılmazken, aksi durumda daha iyi performans için NoSQL tercih edebilirsiniz. NoSQL de **immediate consistency** yerine **eventual consistency** olduğunu söylemiştik.

Biraz açmak gerekirse; Örneğin bir sosyal medya uygulamanız var ve veri tabanı olarak NoSQL'i tercih ettiniz. (doğru tercih)

Trafiğiniz çok olduğundan dolayı servislerinizi bir çok node (sunucu) üzerinde sunuyor, yatayda ölçekleniyorsunuz. Sosyal medya uygulamalarında paylaşılan içeriklere yapılan yorumlar veya beğeni sayılarının, tüm **node**'lar da herhangi bir **t** anında aynı değere sahip **olmaması** büyük bir sorun teşkil etmez. Şöyle ki;

Türkiye'de ki bir kullanıcı, bir paylaşımdaki beğeni sayısını 500 olarak görürken aynı anda aynı paylaşıma bakan Rusya'da ki kullanıcı bunu 495 olarak görebilir. Burada **Eventual Consistency** bir süre sonra sağlanacak ve tüm node'lar da aynı beğeni sayısı tutuluyor olacaktır. Biz bu gibi durumların oluşma ihtimalini bilerek, performans ve yüksek ölçeklenebilirlik için NoSQL tercihi bulunduk ve doğru olanı yaptık.

Immediate Consistency için ise finansal uygulamaları örnek gösterebiliriz. Parasal işlemler söz konusu olduğu için ilişkisel veri tabanlarının daha kararlı **ACID** özelliği önem kazandığından, genelde RDBMS'ler tercih edilmekte.

Örnek Senaryo

Hatırlarsanız, servislerinizin yüklendikleri sorumluluğun boyutlarının doğrudan veri tabanı seçiminiz üzerinde etkisi olduğundan bahsetmiştik. Örnek bir senaryo üzerinden ne demek istediğimize geçmeden önce bir kaç kelam etmekte fayda var.

İdeal Dünya'da bir Mikroservis'in yalnızca tek bir işi yapması ve o işi en iyi şekilde yapması istenir. **SOLID** prensiplerin ilki olan **Single Responsibility** prensibini bilirsiniz, sınıfları ve

metotları mümkün olduğunda atomik tutarak onlara tek bir sorumluluk yüklemeye yöneltir bizi. Mikroservis Mimari’de servisleri tasarlarken bu prensibe bağlı kalmalıyız.

Servisimiz ne kadar büyür ve yaptığı iş ne kadar karmaşıklırsa, RDBMS kullanımı mecburiyet haline gelebilir. O zaman bizde ilişkisel veri tabanı kullanınız diye düşünebilirsiniz. Bu durumda RDBMS ile gelen **JOIN** işlemleri sizi bekliyor demektir. JOIN’leme işlemi maliyetli bir işlemdir esasında. JOIN’lenen tabloların içerdiği veri miktarı arttıkça performansınız yerlerde sürünecektir, öyle ki index’ler bile sizi kurtaramayabilir.

Şimdi, servisleri **mümkün olduğunca** atomik tasarlayarak, NoSQL kullanmak mı? Yoksa ayırabileceğiniz servisleri ayırmayarak RDBMS kullanmak mı? Karar sizin. Bu arada konuyla ilgili Martin Fowler’ın [buradaki](#) yazısına da göz atmanızı tavsiye ederim.

Gelelim örnek senaryomuza. Öncelikle hatalı bir tasarım ile veri tabanı seçimlerini yapacağız. Ardından daha doğrusu nasıl olur sorusunu sorarak, problemli noktaları saptayıp daha doğru bir tasarıma geçeceğiz.

Ön Not: NoSQL / RDBMS seçimini yapmadan önce her servisin yalnızca kendisinin doğrudan erişebildiği izole bir veri tabanı olması gerektiğini bilmelisiniz. Bu izole veri tabanları aynı sunucu üzerinde olabileceği gibi farklı sunucular üzerinde de yer alabilir. Bir servis başka bir servisin verisine ihtiyaç duyuyorsa bu ihtiyacını o servisin veri tabanına doğrudan erişerek **gideremez**. En doğrusu bir **event-bus** üzerinden **full asenkron** bir haberleşme yapılmasıdır. Eğer bu yapılamıyorsa, servisler bir birlerine anlık olarak **http request** yapabilir. Bu anlık http istekleri, servisleri birbirine bağımlı kıldığı için çok doğru bir yöntem değildir ama başka bir servisin veri tabanına direkt erişmek kadar da hayati bir hata değildir en azından.

Uygulama

Bir çok alt firması olan bir holding için ilişkisel veri tabanı kullanan monolith yapıda bir insan kaynakları uygulmamız olsun. Bu uygulamanın bir kullanıcı girişi bir de admin panel tarafı var.

Kullanıcılar sisteme giriş yaparak kişisel bilgilerini girme, izin talep etme, masraf fişi gönderme gibi işlemleri yapabildiği gibi, kalan izin gibi bilgilerini de **read-only** olarak görüntüleye biliyorlar. Sistem adminleri ise, yeni kullanıcı, yeni departman ve yeni firma ekleme gibi işlemleri yönetim panelinden yapabiliyorlar.

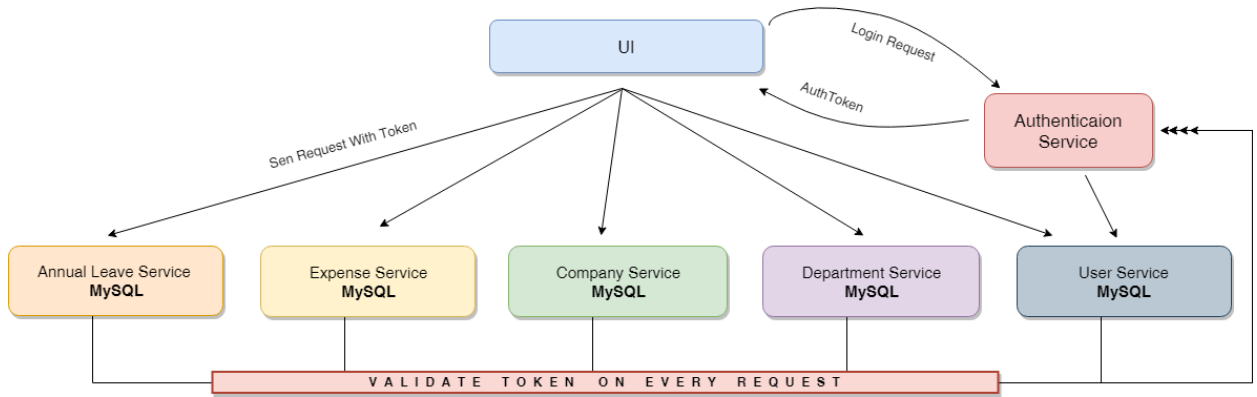
Mikroservis mimariye geiş için yazının girişinde sorduğumuz 3 soruyu sorarak başlıyoruz ve neticede aşağıdaki yanlış tasarımı yaptığımızı düşünelim. (Doğru tasarıma bakmadan yanlışlığın nerede olabileceğini tahmin edebilirsiniz.)

Not: Bu örnek uygulamanın, işlevi ve kullanıcı sayısı göz önüne alındığında, büyük miktarda veri ve ölçeklenme gibi problemleri olmayacağı aşıkardır. Dolayısıyla hatalı tasarım diye belirttiğimiz yapıda kalmasında bir mahsur olmayabilir. Siz bu hatalı ve doğru tasarımların , büyük ölçekli ve yüksek trafikli bir uygulamada yapıldığını hayal edebilirsiniz.

Hatalı Mimari

Monolith uygulamamız MySQL veri tabanına sahip olduğu için Mikroservis Mimari'ye dönüşüm sonrasında tüm servislerimiz için yine MySQL ile devam etmek istedik.

Toplamda 6 adet servisimiz olsun dedik. Bunlardan 5 tanesinin kendisine ait izole bir veri tabanı var. Aşağıdaki gibi çizmeye çalıştım;

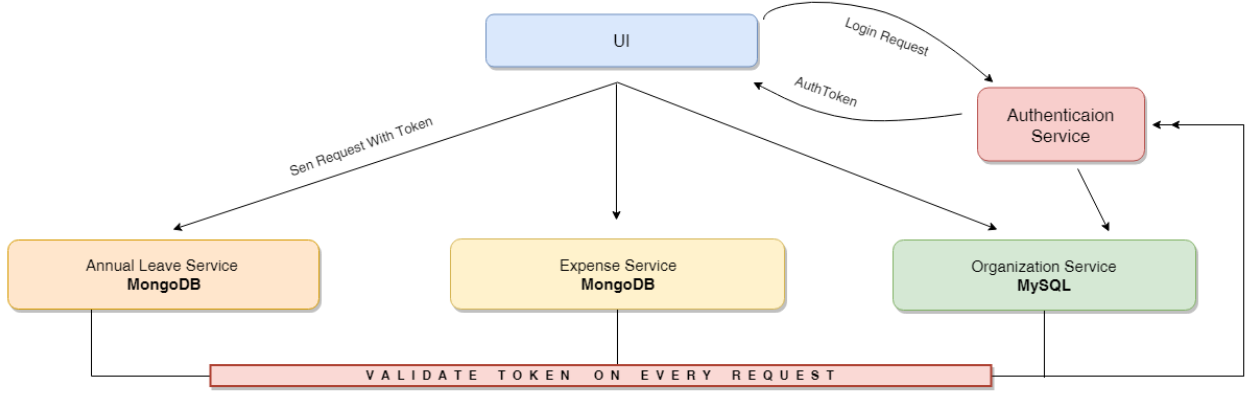


Hatalı Mimari

İyileştirilmiş Mimari

Geliştirmeler ve testler esnasında fark ettik ki, User, Company ve Department servisleri bir birlerine çok fazla http isteği yapıyorlar, yani bağımlılık oranları bir hayli yüksek. Bu durum kullanıcı sayısı arttıkça daha büyük bir sorun haline gelecek gibi duruyor.

Dolayısıyla bu 3 servisi birleştiriyoruz ve yine ilişkisel bir veri tabanı olan MySQL ile devam ediyoruz. Expense (masraf) ve Annual Leave (izin) servisleri ise bağımsız kalmaya devam ediyorlar. Ancak ilişkisel bir veri tabanı kullanmamızı gerektiren bir durum olmadığını düşünerek, MongoDB ile yola devam diyoruz. Mimarının son hali kabaca şöyle şekilleniyor;



Doğru Mimari

Ne Yaptık?

Servis sayımızı azaltarak, 2 servislik bir DevOps yükünden kurtulmuş olduk. Bununla kalmadı tabi, bir birleri arasında çok yoğun bir http trafiği oluşturan 3 servisin meydana getirdiği bu trafiği de bitirmiş olduk. Eğer MySQL yerine ücretli olan MsSQL gibi ücretli bir veri tabanı kullanmış olsaydık ki özellikle ülkemizde çok yaygın bir kullanıma sahip, bu lisans ücretinden de tasarrufa gitmiş olacaktık.

Daha İyisi Olamaz mı?

Her zaman olur. Burada dikkat ettiyseniz servisler bir birlerine doğrudan http call ile erişmekte. Full asenkron ve daha gevşek bağlı bir tasarım için RabbitMQ gibi bir message broker kullanmamız gerekir. Bir önceki, servisler arası iletişim bölümünde detaylı olarak bahsetmiştik hatırlarsanız.

Yanlış Bir Seçim Yaptığımı Nasıl Anlarım?

Her iki tip veri tabanı içinde kısaca ip ucu vermek gerekirse;

Örneğin, Doküman tabanlı NoSQL veri tabanı kullandığınız bir servisiniz de, transaction kullanımına ihtiyaç duyuyorsanız, aynı anda birden fazla dokümanı güncelleme gibi bir ihtiyacınız oluyorsa, NoSQL'in kapsama alanından çıkmış, ilişkisel veri tabanı Dünyasına girmişsiniz demektir.

Benzer şekilde, ilişkisel veri tabanı kullandığınız servisiniz de, tabloları JOIN leme, transaction gibi özellikleri hiç kullanmıyorsanız ve büyük miktarda veri söz konusuysa ki bu durumda yatayda ölçeklenme kaçınılmazdır, o halde NoSQL kullanmak daha doğru bir tercih olacaktır.

Son olarak;

NoSQL veri tabanlarını RDBMS'lere ucuz bir alternatif olarak düşünerek kullanmak hata olur. Ne olduklarını ve hangi yaraya merhem olduklarını bilerek, bilinçli bir seçim yapmak zorundayız.

Servisler Arası İletişim

Bu bölümde, “Servisler arasındaki haberleşmeyi nasıl sağlamalıyız?” sorusuna aşağıdaki 3 başlık altında yanıt arayacağız.

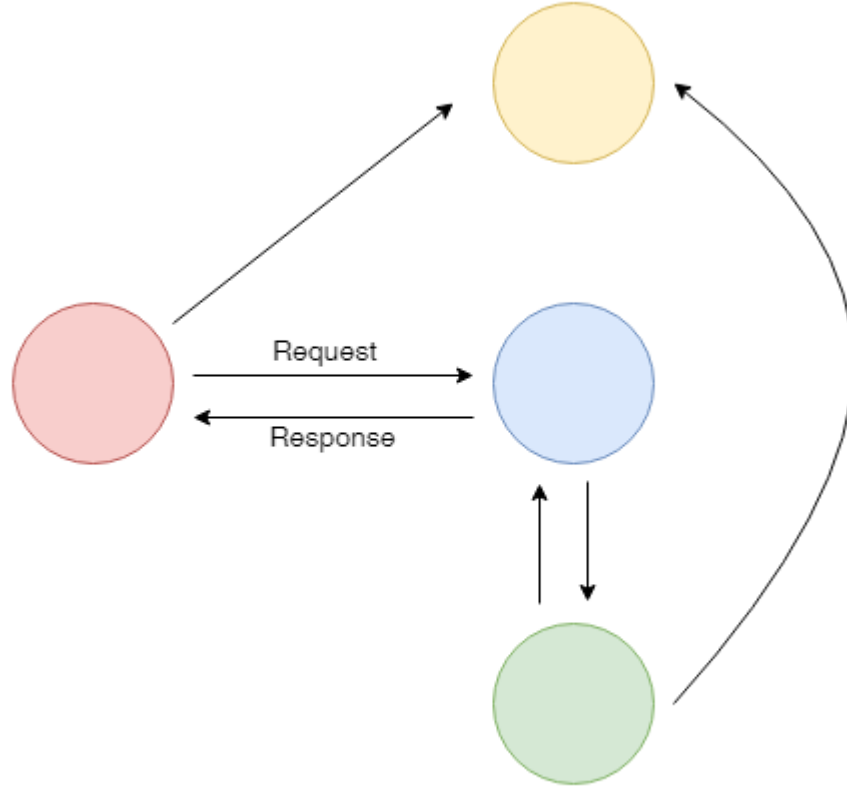
- Request-Driven Mimari
- Event-Driven Mimari
- Hybrid Mimari

Request-Driven Mimari

Servis sayımız arttıkça mimarimizin karmaşıklığı da, http trafiğimiz de doğru orantılı olarak artacaktır. Bir Mikroservis , verisine ihtiyaç duyduğu başka bir servise, bir http client üzerinden istekte bulunur ve bu işlem bir **IO** (Thread Blocking IO) işlemi olduğu için aslında maliyetli de bir işlemdir. Servisler (sunucular) client’larıyla olan bu iletişimi **socket**’ler üzerinden yaparlar ve bu socket’ler sonsuz sayıda değildir, bu yüzden socket kullanımı iyi yönetilmesi gereken bir konudur.

Dolayısıyla mimarimizi tasarlarken servislerimizin kendi aralarında yaptıkları isteklerin sayısı konusunda biraz cimri davranmamız gerekiyor. Tabi bunu yaparken, aslında ayrıştırılması gereken servisleri sırf http istek sayısını azaltacağız diyerek birleştirmek de yanlış olacaktır. Aşağı tükürsek sakal yukarı tükürsek bıyık durumu yani. Mikroservis Mimari’nin zorluklarının üstesinden gelmek de bu gibi kararları doğru verebilmekten geçiyor zaten.

Request-Driven Mimari



Request-Driven Mimari Örnek Çizim

Bu mimari de adından da anlaşılacağı üzere bir service, verisine ihtiyaç duyduğu bir diğer servise doğrudan istekte bulunur. Servislerimizin modern **Rest** servisleri olduğunu kabul edersek yapılan istekler, **GET, POST, DELETE ve UPDATE** isteklerinden ibaret olacaktır.

Yukarıdaki çizimde **fire-and-forget** iletişim şeklini de göstermiş olmak adına, sarı renkli olan servisten geri dönüş belirtmedim. Elbette her http request'in bir http response'u olacaktır, burada isteği yapan kırmızı ve yeşil servislerin dönen resonse ile ilgilenmediklerini anlıyoruz. Örneğin bu sarı renkli servisimiz notifikasyon veya log servisimiz olabilir. (Loglama işlemini http servis üzerinden yapmak **genelde** yanlış bir tercih olur)

Request-Driven Mimari'de servislerimiz arasındaki iletişimi iki yolla sağlayabiliriz;

Senkron (Synchronous) İletişim

Http protokolü senkron çalışan bir protokoldür. Client bir istek yapar ve sunucudan yanıt dönmesini bekler. Client tarafında servis çağrısını yapan kodun senkron (thread'in

blocklanması durumu) veya asenkron (thread'in bloklanmaması ve yanıtın **call back** ile gelmesi) yazılması Http'nin senkron bir protokol olduğu gerçeğini deęiřtirmez.

Burada aslında ilginç bir durum söz konusudur. Client'in servis çağrısını asenkron olarak yapması(**call back** yapısı ile) veya sunucunun yanıtı asenkron olarak dönmesiyle aslında bir anlamda senkron bir protokol olan Http'den asenkron yanıt vermiş/almış oluyoruz. Ancak bu Http protokolünü deęiřtirdiğimiz onu asenkron bir şekilde çalıştırdığımız anlamına gelmiyor elbette.

Buna örnek olarak .Net Framework 4.5 ile gelen **async/await** yapısını verebiliriz. Bu yapı, asenkron istek yapma veya yanıt dönme işlemlerini çok kolay bir şekilde yapmamıza imkan sağlıyor. Dolayısıyla Mikroservis'lerinizi geliřtirdiğiniz framework'ün ve dilin buna benzer bir yapıya sahip olması, yani isteklerinizi ve yanıtlarınızı asenkron olarak yapmanıza imkan sağlaması önem arz ediyor.

DotNet'in yanı sıra dięer popüler framework'lerin (java, php, nodejs, ruby, go vs.) hepsi benzer bir call back yapısına sahip midir, yüksek ihtimalle evet. Ancak dotnet'in **async/await** yapısının kodu sadeleřtirdięi kadar sadeleřtirebilirler mi emin deęilim açıkçası. Burada biraz dotnet güzellemesi yapmış olduk ama bana göre **async/await** yapısı bunu fazlasıyla hak ediyor.

Asenkron (Asynchronous) İletişim

Http'nin senkron çalıştıęından ve call back mekanizmalarıyla bir şekilde client veya sunucu tarafında bir **asenkronezasyon** elde ettiğimizden bahsettik.

Servisler arası iletişim için Http haricinde kullanabileceğimiz, üstelik asenkron bir protokol olan **AMQP**'den (Advanced Message Queuing Protocol) kısaca bahsedelim.

Wikipedia tanımı;

The **Advanced Message Queuing Protocol (AMQP)** is an open standard application layer protocol for message-oriented middleware.

AMQP'nin en önemli özelliklerini, mesaj yönlendirme, kuyruklama, routing (p2p ve pub/sub), dayanıklılık (güvenilirlik) ve güvenlik olarak sıralayabiliriz. AMQP, çok farklı yapıda ve

birbirinden bağımsız çalışan sistemler arası iletişimi kolaylaştırdığı için, sistemlerin birlikte çalışabilirlik (**interoperability**) yönlerini güçlendirmemize de olanak sağlıyor.

AMQP'nin reliability (güvenilirlik) özelliğine ayrı bir parantez açmak gerekiyor. AMQP bu özelliğini, içerdği 3 farklı teslimat garanti (**delivery guarantees**) modu ile kazanmıştır. Kısaca açıklamak gerekirse;

- **at-most-once** : Publisher mesajı **en fazla** 1 kere gönderir ve bu yöntemde mesajın kaçırılma riski vardır çünkü consumer'dan mesajı aldığına dair bir teyit beklenmez. RabbitMQ ve Kafka destekler.
- **at-least-once** : Publisher mesajı **en az** 1 kere gönderir ve bu yöntemde mesajın tekrarlı (duplicate) gönderilme riski vardır çünkü consumer'dan teyit alınırken bir hata meydana gelebilir. RabbitMQ ve Kafka destekler.
- **exactly-once** : Publisher'ın mesajı bir veya yalnız bir kere göndermiş olmasını garanti eder. Diğer iki yöntem kadar yaygın değildir. Daha spesifik, yani kısıtlı senaryolar için kullanışlı olabilir. Yalnızca Kafka destekler. **exactly-once** için daha fazla bilgiye [buradan](#) erişebilirsiniz.

Sisteminizin bu 3 yöntemden hangisini tolere edeceğine siz karar vererek uygun konfigrasyonu yapmalısınız. Örneğin, işlemleriniz **idempotent** yapıdaysa ve dolayısıyla bir mesajın kuyruktan 2 kere alınıp işlenmesi durumu sizin için sorun teşkil etmiyorsa **at-least-once** modunu seçebilirsiniz.

RabbitMQ, **AMQP**'yi destekleyen modern message broker'lara baktığımızda ilk aklımıza gelenlerden. Öyle ki, AMQP hakkında araştırma yaptıysanız karşınıza sürekli RabbitMQ'nün çıktığını görmüşsünüzdür. RabbitMQ hali hazırda MQTT, STOMP, AMQP gibi bir çok mesajlaşma protocol'ünü desteklemektedir. Sonraki bölümlerde RabbitMQ'den biraz daha bahsedeceğiz.

Event-Driven Mimari

Mikroservis Mimari dünyasında en zor konulardan birisi data bütünlüğünün sağlanması ve herhangi bir **t** anında tüm data'mızın anlamlı yani beklenen bir **state**'de olmasının garanti edilmesi konusudur. Buna transaction bütünlüğünün sağlanması da diyebiliriz. Monolith mimariye göre kıyas götürmeyecek kadar zorlu bir süreç olduğu herkesin malumu.

Bir önceki transaction yönetimi bölümünde bu konuyu daha detaylı olarak incelemiştik, burada ise transaction yönetimi konusunun bizi **event-driven** mimariye götüren yanlarına değineceğiz.

Event Derken?

Bu mimari'de event ile kastettiğimiz şey, bir servisin kendi domain'inde bir kaynağın durumunu değiştirmesiyle birlikte bu değişiklik bilgisini ilgili servis veya servislerle paylaşmasıdır. Bu paylaşımı da genelde bir mesaj kuyruk yapısı üzerinden yapar. Sürekli duyduğumuz **event fırlatma** tabirinin bu mimarideki karşılığı "ben şu değişikliği yaptım, onu da şu kuyruğa veya kuyruklara yazdım ilgililere duyurulur" demektir.

Event-Driven mimariye hiçte yabancı değiliz aslında, gündelik hayatımız da bile bazı örnek senaryolar mevcut. Örneğin; canınız hamburger çekti ve soluğu bir hamburgercide aldınız ve sipariş için sizi ilk karşılayan görevliyle konuşuyorsunuz, siparişinizin detayını verdiniz ve beklemeye başladınız.

Siparişiniz hazır olana dek mecburen bekleyeceksiniz. Peki siparişinizi alan görevli? Sizinle beraber boş boş bekleyip, siz siparişinizi teslim aldıktan sonra sıradaki müşteriyle ilgilense nasıl olurdu? Elbette saçma olurdu, ancak **senkron** çalışan **request-driven** mimaride olan şey tam da bu.

Ama biz asenkron iletişim istiyoruz, kaynaklarımızı daha verimli kullanmak ve daha iyi performans için buna ihtiyacımız var. Devam edelim.

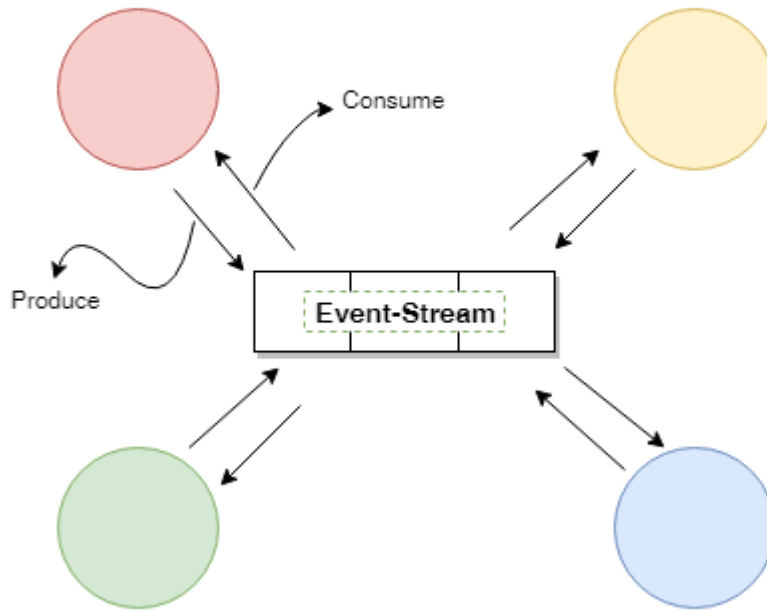
Siparişi sizden alan görevli bunu siparişi hazırlayacak olan kişiye iletir. (**order_created event'i**) Ardından sıradaki müşteriyle ilgilenirken, siz boş bir masa bulup alarım sizin için çalmasını beklersiniz. Siparişiniz bitince ilgili çalışan "x nolu sipariş hazır" bilgisini verir, (**order_ready event'i**) ve sıradaki siparişi hazırlamaya başlar. **Order_ready** event'inin muhatabı olan çalışan size siparişinizin hazır olduğunu bildirir ve siz gider teslim alırsınız. (**order_delivered event'i**) Çok uzatmamak adına bazı detayları atlamakla birlikte, gündelik hayattan basit bir **event-driven** süreç örneği vermiş olduk aslında.

Aşağıdaki örnek çizimde 4 adet Mikroservis ve bir mesaj kuyruğumuz mevcut. Buradaki okların tamamı iki yönlü olmak zorunda değildi. İki yönlü olması, o servisin aslında hem bir **producer** yani event oluşturan, hem de bir **consumer** yani event dinleyen (tüketen)

olduğunu gösteriyor bize. Bir servis sadece producer, sadece consumer veya her ikisi de olabilir üstlendiği göreve göre.

Örneğin; örnek senaryomuzda hamburgeri hazırlayıp diğer görevliye teslim eden görevli, **order_created event**'ini dinleyerek, **order_ready event**'ini oluşturduğu için iki yönlü bir servistir diyebiliriz.

Event-Driven Mimari



Event-Driven Mimari Örnek Çizim

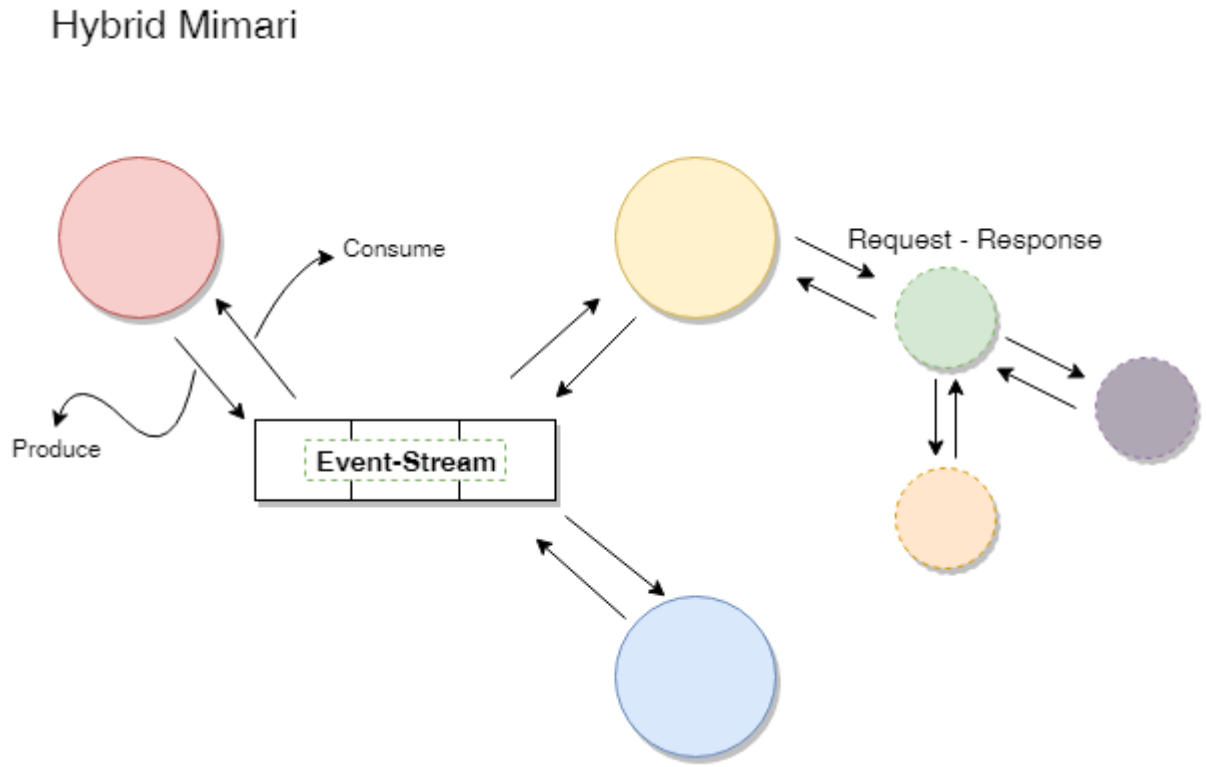
Event-Driven mimarinin avantajları olarak şunları sayabiliriz;

- Gevşek-Bağlı(loosely coupled) bir mimari oluşturmamıza olanak sağlar.
- Gevşek bağıllık servislerin **development** eforunu azaltır.
- İletişimimiz asenkron olacağı için performans kazanımı sağlar.
- Yatayda kolay ölçeklenebilirlik (scalability) sağlar.
- Mesaj kuyruğu sayesinde, consumer servis bir sebepten ötürü erişilemez durumdayken veri kuyrukta kalır ve tekrar erişilebilir olunca data kaybı yaşamadan kuyruğu tüketmeye devam eder. (Sonraki bölümde bahsedeceğimiz notifikasyon servis örneği)

Four years from now, “mere mortals” will begin to adopt an event-driven architecture (EDA) for the sort of complex event processing that has been attempted only by software gurus building operating systems or systems management tools, and sophisticated developers at financial institutions, predicted Roy Schulte, an analyst at Stamford.

Hybrid Mimari

Bu mimari adından da anlaşılacağı üzere bahsettiğimiz iki mimarinin birlikte kullanıldığı yapıları ifade eder. Eğer tamamen event-driven mimari ile kurulan bir sisteme sahipseniz veya en azından aşinalığınız varsa, “hybrid yapıya ne zaman ihtiyaç duyulur ki” diye düşünebilirsiniz. İlk bakışta gerçek hayat örneklendirmesi yapamayabilirsiniz doğal olarak. Neticede event-driven yaklaşım, request-driven kadar yaygın değil. En azından benim gördüğüm kadarıyla öyle. Bu yüzden genelde, request-driven ile inşa edilen sistemlere ihtiyaca göre bir **event-bus** sonradan entegre edilebiliyor.



Hybrid Mimari Örnek Çizim

Bir gerek hayat rneęi verecek olursak;

Request-Driven mimariyle kurgulanmıř Mikro servis'leriniz var. Uygulamanıza notifikasyon gnderme zellięini ekleyeceksiniz. eřitli servisler kendi domain'lerinin gerektirdięi řekilde kullanıcılara doęrudan bazı notifikasyonlar gndermelidirler. Hemen Notification Service adında yeni bir Mikro servis oluřturdunuz ve request-driven yapınıza entegre ettiniz. Bu řekilde devam edebilirsiniz elbette.

Peki burada notification service kendisine gelen notifikasyon gnderim isteklerini bir kuyruktan okuyamaz mı? Dięer servisler RabbitMQ, Kafka gibi bir kuyruk yapısı zerinden mesajları notification service'in dinledięi bi kuyruęa gnderebilir ve iřlerine devam ederler. **Fire-and-Forget** dedięimiz iletiřim řekli yani. Notification service ise notifikasyon isteklerini gnderen servislerle doęrudan iletiřim ierisinde deęildir artık ve isteęin kimden geldięini de bilmek zorunda deęildir. Artık tek yapması gereken bir kuyruęu dinlemek ve gelen datayı alarak notifikasyon gnderme iřini icra etmektir.

Bu řekilde tasarlayarak baęımlılıklarımızı azaltmıř olduk. Bir dięer avantajı ise, notification service'imiz bir sebepten tr erişilemez hale gelirse, iletiřimi asenkron hale getirdięimiz iin servis tekrardan erişilebilir olduktan sonra kuyrukta birikmiř olan veriyi alarak notifikasyonları eksiksiz olarak gndermeye devam edecektir. Aynı senaryoda Request-Driven mimaride eęer bir **retry-policy** uygulamadıysak notifikasyonları kaybetmiř olacaktık ki, **retry-policy** uygulamamıza raęmen kaybetme ihtimali yine olabilir.

Bir sonraki blmde servislerimizin bir birlerine yaptıkları http isteklerinden tamamen kurtularak, daha baęımsız servisler elde edebilmemize imkan saęlayan bir pattern'dan ve Domain-Driven Design'dan bahsedeceęiz.

DDD ve Mikroservis Mimari

Bir önceki bölümde bahsettiğimiz iletişim yöntemlerinden olan asenkron iletişimde event-based mimari uygulansa bile servisler veri paylaşımı için birbirlerine anlık olarak http istekleri attıkları için "bağımsızlık" prensibine aykırı bir durum olduğundan kısaca bahsetmiştik. Bu bölümde bu http isteklerinden de kurtularak daha izole ve bağımsız servislere nasıl ulaşırız sorusunun yanıtını arayacağız.

Konuyu aşağıdaki alt başlıklarda ele alarak anlatmaya çalışacağım. Bu bölümde esas değinmek istediğim konu son madde, yani veri paylaşımı konusu. Bu önemli konuyu örnek bir senaryo üzerinden inceleyeceğiz. Önceki bölümleri bir ön hazırlık olması açısından eklemenin iyi olacağını düşündüm.

- Mikroservis Mimari ve Domain-Driven Design
- Bounded Context nedir?
- Her Bounded Context bir Mikroservis anlamına gelir mi?
- **Bounded Context'ler (Mikroservis'ler) arası veri paylaşımı**

Mikroservis Mimari & DDD

Mikroservis Mimari ve DDD gibi iki ağır konuyu bir başlık altına sığdırmaya çalışmayacağız elbette, ki biraz iddialı bir hedef olurdu bu.

Bu bölümde, bu iki mimari arasındaki ilişkiyi birlikte kullanılabilirlik yönünden inceleyemeye çalışacağız.

- **Birlikte Kullanmaktan Kastımız Nedir?**

Ekip olarak yeni ve uzun soluklu bir projeye başlıyorsunuz ve proje belirli bir olgunluk seviyesine ulaştıktan sonra monolith den Mikroservis mimariye dönüştürmeyi planlıyorsunuz diyelim. Hedefte kesin olarak Mikroservis mimari olmasına rağmen, başlarken monolith yapıda başlayarak devam ediyorsunuz ki bence de böyle yapmalısınız. (Monolith mi yoksa Mikroservis mi başlamanın daha doğru olduğu sorusu bu yazsının konusu olmadığından burada üzerinde durmadan devam edelim.)

Monolith yapınızı kurgularken aldığınız teknik kararların, ileride Mikroservis Mimari'ye dönüşüm sürecinizin zorluk seviyesini belirleyeceğini unutmamalısınız. Tam bu noktada DDD'den bahsedebiliriz. Monolith yapıda başladığımız projede DDD'yi prensiplerine sadık kalarak doğru bir şekilde uygularsanız, bu dönüşüm işlemini hem daha kolay hem de daha doğru ve daha az taviz vererek yapabilirsiniz. Bu avantajı bize sağlayacak olan ve bir sonraki bölümde bahsedeceğimiz kavram DDD'nin Bounded Context kavramı.

Bu arada, monolith mimarinizde DDD uygulamazsanız Mikroservis Mimari dönüşümü yapılamaz gibi bir mesaj vermeye çalışmıyorum. Amacımız gevşek bağlı bir mimari oluşturmak ve DDD'nin de burada bize yardımcı olabileceğinden bahsediyorum aslında. Yani DDD buradaki tek alternatifimizi değil elbette.

Mesela, son zamanlarda adını biraz daha fazla duymaya başladığım **Modular Monolith** tasarımdan da çok kısa bahsetmek isterim. Aslında bu mimari için ismiyle müsemma demek yanlış olmayacaktır. Aşına olduğumuz monolith mimarinin modüler, yani bağımlılıklardan mümkün olduğunca arındırılmış bir tarzda kurgulanması diyebiliriz. Konuyla ilgili [burada](#) güzel bir makale mevcut bir göz atmanızı tavsiye ederim. Yine aynı makale yazarının, DDD ve Moduler Monolith mimariyi birlikte uygulayarak geliştirdiği, incelemeye değer gördüğüm bir projeye de [buradan](#) ulaşabilirsiniz.

Bounded Context Nedir?

Bounded Context, DDD'nin anlaşılması biraz zaman alan ve aynı zamanda da en önemli kavramlarından birisidir. Burada “zor” ile kastettiğim şey; **Bir Bounded Context'in sınırlarının belirlenmesi** konusu. Aynı zorluk bir **Aggregate**'in tanımlanması için de geçerli diyebilirim.

Domain'im de kaç tane Bounded Context olduğu, bunların sınırları ve birbirleriyle hangi noktalarda ilişkili oldukları konuları kritik önem arz ediyor.

- **Bounded Context'leri Keşfedilmesi**

Bounded Context'leri domain expert'ler ile konuşarak ve bazı ip uçlarından faydalanarak ortaya çıkarmalıyız. Bu ip uçlarına gelmeden önce belirtmekte fayda var; Bounded Context leri bir kere belirledim, artık değişmez gibi bir düşünceye kapılmamalıyız. Domain

expert'lerle konuşarak ve değişen şartları da göz önüne alarak çizdiğiniz sınırlarda değişiklikler yapmanız, Bounded Context'lerinizi yeniden şekillendirmeniz gerekecektir.

(Hatta belki bu sınırlar belirginleşene kadar DDD'yi bir kenara bırakmalı ve Bounded Context'lerinizi büyük ölçüde netleştirdikten sonra uygulamanızı DDD için refactor etmelisiniz. Ancak bu refactoring sürecinin maliyeti sizin bu süreci başlatma zamanınıza göre belirleneceği için elinizi çabuk tutmanız gerekebilir.)

Bir Bounded Context'in sınırlarını belirlerken domain expert'ler ile doğrudan konuşma dışında hangi ip uçlarından faydalanabiliriz biraz bunlara değinelim.

Kullanıldığı domain'e göre farklı anlamlara gelebilen kavramları bulmaya çalışın. Örneğin 'x' kelimesi kullanıldığı yere göre 2 farklı anlama bürünüyorsa, bu 2 farklı Bounded Context'in varlığını işaret ediyor **olabilir**. Örneğin; **Product** kelimesi, **Shipment Context**'in de ağırlığı olan taşınacak bir **yük** anlamına gelirken, **Inventory Context**'in de elde kaç adet bulunduğu veya mevcut olup olmadığı önemli olan bir **sayı**dan ibaret aslında.

Diğer bir ip ucu olarak; Bounded Context'lerinizi tanımladınız ve geliştirme süreciniz devam ediyor diyelim. Ancak bir sorun var, bir context'de ki bir veriyi değiştirdiğinizde başka bir context'te de bir veri değiştirmek zorunda kalıyorsunuz. Bu iki context'in birbirine bağımlı olduğu anlamına geliyor. Daha da kötüsü bu durum farklı farklı veriler için sıkça yaşanmaya başlıyor. Bu durumda, bu ikisi context'in tek bir context altında birleşmesi durumunu değerlendirmemiz gerekiyor.

Son olarak, DDD'de yer alan Aggregate Root, Entity, Value object, Domain Event gibi tanımlamaları en doğru şekilde yapabilmemiz için, Bounded Context'lerimizi de en doğru şekilde tanımlamamız gerektiğini bilmemiz gerekiyor.

- **Neden 'Bounded' Context?**

Eric Evans'ın [kitabının](#) kapağında da belirttiği gibi, DDD'nin karmaşık domainlerde, yazılımın merkezinde yer alan o karmaşıklıkla mücadele ettiğini biliyoruz.

Yazılımda karmaşıklık ve bu karmaşıklıkla başa çıkabilme konularına baktığınızda önünüze ilk çıkan şeylerden birisi **loosely coupled (gevşek bağlı)** bir tasarımın gerekliliği olur. Burada birbirine bağlı olmayan veya çok az bağımlı olmasını istediğimiz bu parçacıkları genelde **module** olarak isimlendiriyoruz. Modüler tasarım ifadesini çokça duymuşsunuzdur. DDD'de

birbirine bağımlı olmaması gereken bu modüller **bounded** yani sınırlı context'ler olarak isimlendiriliyor.

Gerçek dünyada domain'lerin kesin hatlarla belirli olmayan sınırları vardır. Bazı noktalarda benzerlik gösteren, ortak kavramlar barındıran domain'ler olabilir. Yukarıda bahsettiğimiz **Product** kavramının hem **Product** hem de **Shipment** context leri için anlamlı olması örneğindeki gibi, Product ve Shipment domain lerini kesin hatlarla ayıramıyoruz.

Ancak yazılım Dünyasında bu gevşek bağlı mimariyi kurgulayabilmemiz için sınırları kesin olarak belirli olan modüllere ihtiyacımız vardır. DDD ye göre, Product kelimesinin Product Context'inde ki anlamıyla Shipment Context'indeki anlamı farklıdır. Yani hangi context sınırları içerisindeyse ona göre anlam kazanır. **Bounded** ifadesinin bu duruma vurgu yapmak için kullanıldığını düşünüyorum.

Bir Bounded Context == Bir Mikroservis ?

Bu sorunun her koşulda doğru olan bir cevabı yoktur diyebiliriz. (Yazılım mimarilerinde birçok konuda olduğu gibi)

Bir Mikroservis, bir Bounded Context'i veya onun bir bölümünü temsil edebilir. Diğer bir deyişle, bir Bounded Context birden fazla Mikroservis'te doğurabilir. Bu tamamen, söz konusu Mikroservis'in ölçeklenebilme ve bağımsız hareket edebilme gereksinimine bağlı olarak verilecek bir karardır aslında. Bunlar esasında birbirine benzer iki kavram olmakla beraber ;

Bir Bounded Context bize domain'in sınırlarını çizerken, bir Mikroservis, domain'den etkilenmekle beraber teknik ve organizasyonel sınırları belirler.

Özetlersek, DDD ile geliştirdiğimiz monolith uygulamamızın Mikroservis Mimari'ye dönüşümü yaparken, "Her Bounded Context için bir ve yalnızca bir Mikroservis oluşturmalıyız" gibi bir kalıbın içine girmemiz yanlış olacaktır diyebiliriz.

Bounded Context'ler (Mikroservis'ler) Arası Veri Paylaşımı

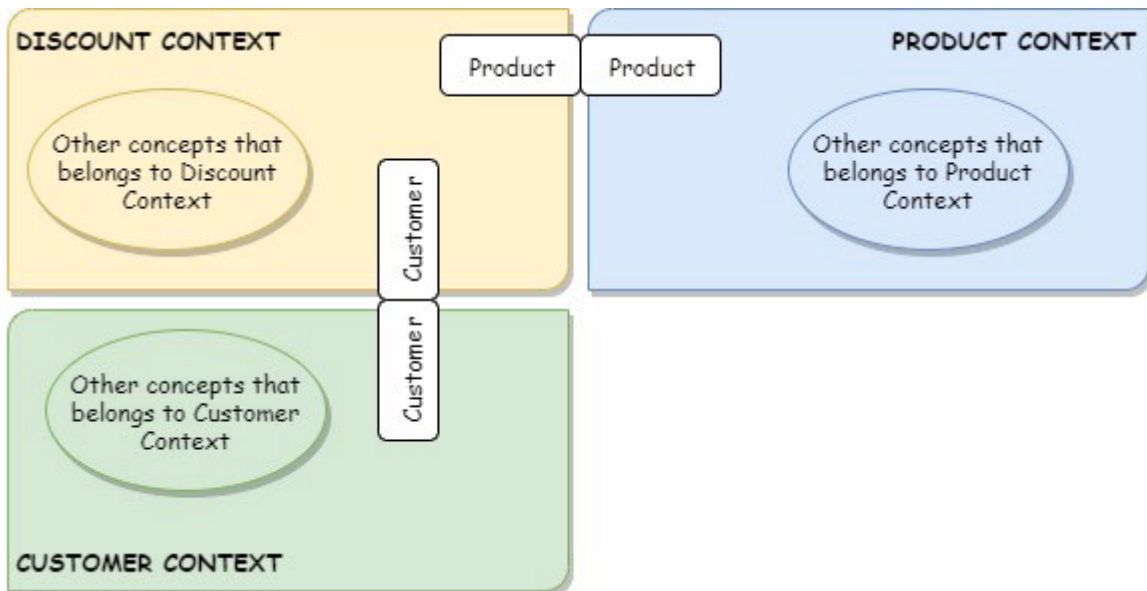
Evet geldik bu bölümde esas bahsetmek istediğimiz konuya. Buraya kadar olan kısım aslında bu bölüme bir ön hazırlıktı diyebiliriz . Başlamadan önce buradaki esas amacımızı tek cümleyle özetlemek gerekirse ;

Birbirlerinin verisine ihtiyaç duyan servislerimizin bu ihtiyacını, servisleri birbirlerine bağımlı hale getirmeden, sınırları(boundaries) ihlal etmeden giderebilmek, şeklinde ifade edebiliriz.

Bu bölümde 3 örnek Bounded Context arasında ki ilişkili noktaları ve bu ilişkinin getirdiği veri paylaşımı zorunluluğunu asenkron olarak çözmemizi sağlayan bir yöntemden bahsedeceğim. Bu yöntemle alakalı Julie Lerman'ın [buradaki](#) yazısını incelemenizi tavsiye ederim.

Meseleyi 3 adet **basitleştirilmiş** Bounded Context üzerinden ele alalım. Bunlar **Customer**, **Product** ve **Discount** context'leri olsun. Customer müşteri ile alakalı iş kurallarını içerirken, Product ürün bilgileri ve Discount satılan ürünler için indirim uygulama iş kurallarını içeriyor.

Bounded Context'lerimizi ve birbiriyle ilişkili oldukları noktaları aşağıdaki gibi göstermeye çalıştım.



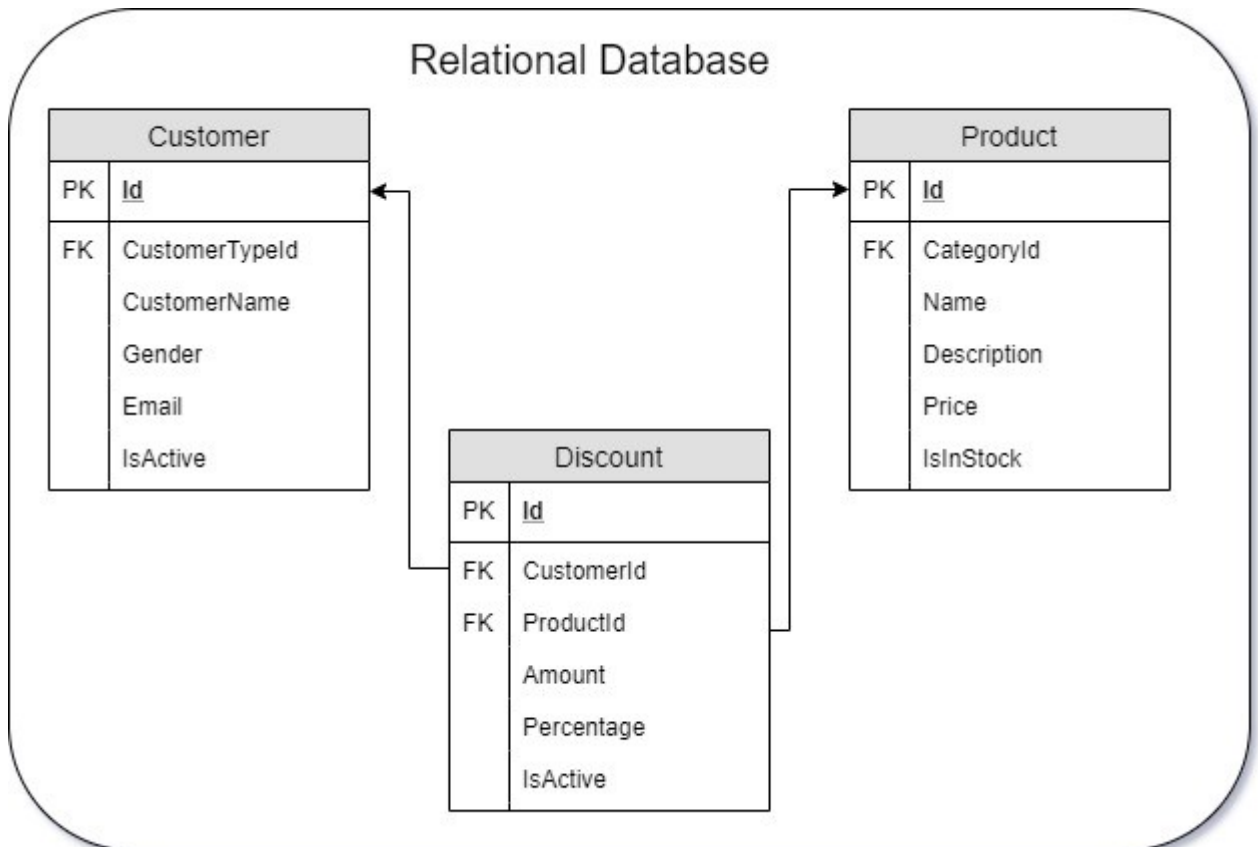
Örnek Bounded Context'ler

Dikkat ettiyseniz Discount context'i hem Product hem de Customer ile ilişkili durumda. Bir diğer deyişle Discount bu iki context in verisine ihtiyaç duymakta. Yuvarlak içerisinde belirttiğim diğer konseptler ise sadece o context içerisinde bir anlamı olan ve diğer context leri ilgilendirmeyen entity'ler.

Bir örnek vermek gerekirse, Product context i içerisinde **ProductCategory** adında bir entity daha var. Discount servisi indirim uygularken ürünün kategori bilgisine de ihtiyaç duysaydı bu category entity sini de ilişkili olarak göstermemiz gerekecekti. Ancak bu örnek senaryomuzda Discount servisinin müşterinin tipine göre bir ürüne indirim uygulaması isteniyor.

Örneğin bazı premium müşterilere, aynı gün yaptıkları ikinci alışverişte %50 indirim uygulanması gibi bir iş kuralımızın olduğunu düşünelim. Bu durumda Discount servisimiz, hem ürünün **Id** ve **Price** bilgisine hem de müşterinin **Id** ve **CustomerType** bilgisine ihtiyaç duymaktadır. Bu 4 bilgi haricindeki diğer bilgilerle ilgilenmediğini vurgulayarak devam edelim.

Eğer monolith yapıda ve bir tek ilişkisel veri tabanına sahip bir uygulamamız olsaydı kabaca aşağıdaki gibi tasarlayabilirdik.



Monolith Mimari İçin İlişkisel Veri Tabanı

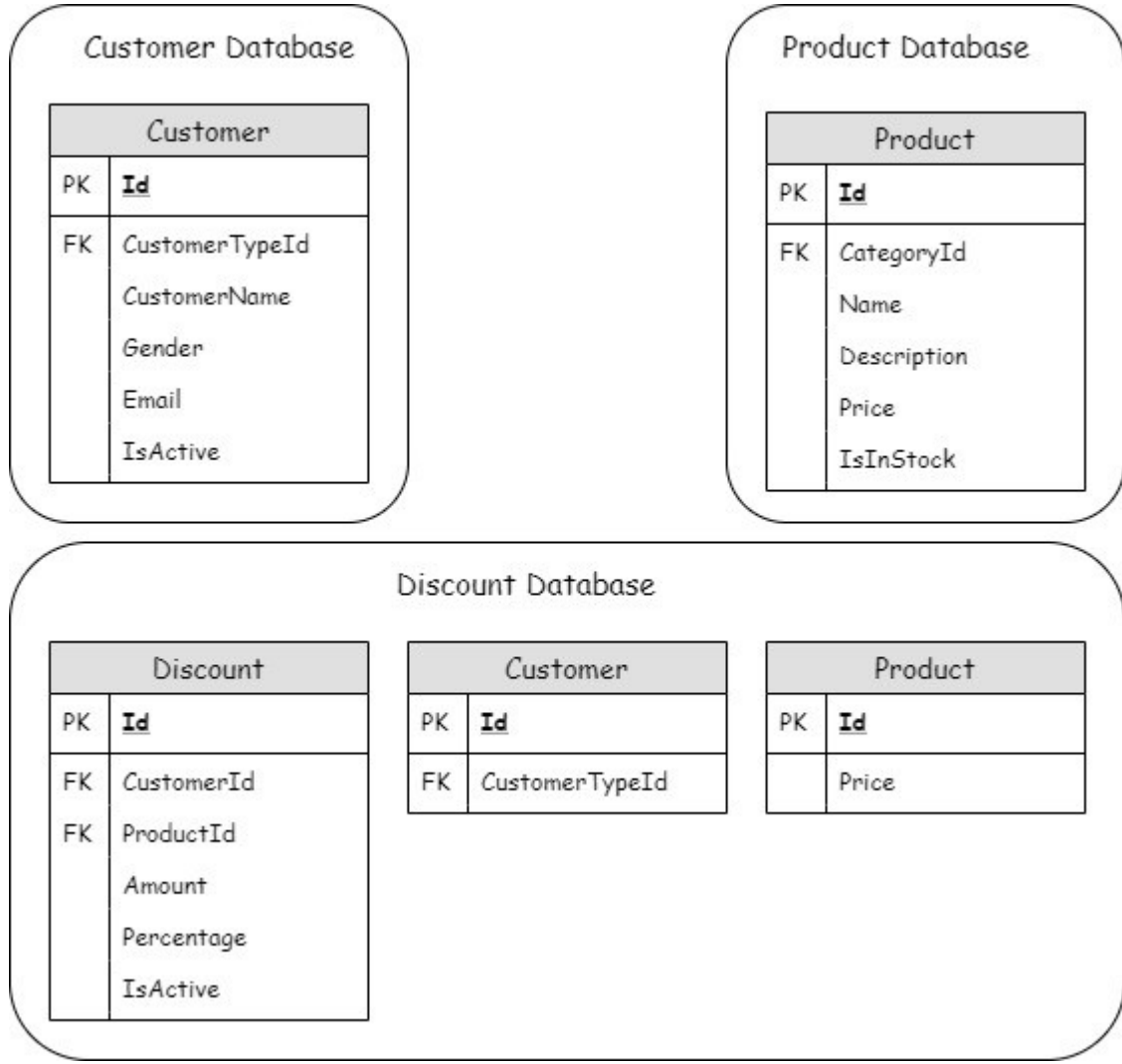
Discount işlemi uygulanırken Discount tablosuna, “X ürünü için Y müşterisine tanımlı bir indirim var mı?” sorgusuyla gelerek süreci yönetebiliriz.

DDD’ye geri dönersek, biz context’lerimizi birbirinden izole ederek otonom bir yapıya bürünmelerini istiyoruz ve aradaki iletişimin event-based, yani asenkron olmasını istiyoruz. Peki bu durumda Discount servisi indirim uygularken Customer ve Product verisine ihtiyaç duyduğu anda nasıl bir yol izlemeli? CustomerId’yi kullanarak CustomerType bilgisine nasıl ulaşmalı? Customer Service’e, http isteği yaparak bu ihtiyacını pek tabi giderebilir ancak daha öncede söylediğimiz gibi bu servislerimizi birbirine bağımlı hale getirdiğinden biz bunu istemiyoruz. (Örneğin, [buradaki](#) 40. satırda oluşan bağımlılık gibi.)

Peki ne yapmak lazım?

Discount servis, Product ve Customer servislerinden sadece ihtiyacı olan verilerin **read-only** bir kopyasını kendi veri tabanında saklarsa nasıl olur? Buna uygun olan yeni veri tabanı yapımız aşağıdaki gibi şekillenecektir. Dikkat ederseniz bir ilişkisel veri tabanımız varken artık 3 izole veri tabanına sahibiz. Bu veri tabanları sql/nosql/graph vb. her hangi bir tipte olabilir. Bu şuan konumuz dışında.

Not: DDD’de her Bounded Context için ayrı ve izole bir veri tabanı olması şartı yoktur. Aynı veri tabanında şema bazlı bir ayırma da gidilebilir. (Product.Product, Discount.Product, Discount.Customer gibi.)



Her Mikroservis İçin İzole Veri Tabanları

Discount servisin veri tabanında customer ve product tablolarının read-only kopyalarının olması gerektiğini belirtmiştik. Burada read-only den kastımızı biraz açalım.

Sisteme yeni bir ürün eklendiğinde **Product** service, **ProductCreated** Domain Event'ini fırlatır. Bu event'i dinleyen **Discount Service** (discount service yerine sadece bu işi yapmakla sorumlu başka bir service olması daha doğru olur) event'i yakalayarak **Discount** servisin veri tabanındaki **Product** tablosuna bu ürünü ekler. Ancak dikkat ettiyseniz bu tabloda sadece 2 alan mevcut, **Id** ve **Price**. Daha önce belirttiğimiz gibi burada tüm product verisini tutmamıza gerek yok, sadece Discount Context'i için anlamlı olan ürün verisini saklıyoruz.

Bu product tablosuna event harici başka bir yolla veri yazma ve silme işlemi kesinlikle yapılmamalıdır. Yani, **ProductCreated**, **ProductDeleted**, **ProductUpdated**, **ProductDeactivated** vb. gibi Domain Event'lerin oluşması haricinde hiçbir şekilde bu tablo üzerinde bir değişiklik yapılmamalıdır diyebiliriz. Read-only den kastımız buydu aslında.

Eğer bu yöntemi ilk kez duyduysanız şuan kendinize şunu soruyor olmalısınız; “Discount veri tabanında ki bu Product ve Customer kopya tablolarının esas veri kaynağıyla olan senkronizasyonundan nasıl emin olacağız? Başımıza iş almıyor muyuz?” Evet alıyoruz aslında.

Sisteme yeni bir ürün eklendiğinde bu ürün discount service’in Product tablosuna eklenemezse ne olacak? Veri tabanına yazma işlemi sırasında bir hata meydana gelebileceği gibi, ilgili event bus’a hiç gönderilememiş bile olabilir. Event-Driven mimari ile uğraştıysanız kaybolmuş, akıbeti meçhul olan event problemiyle karşılaşmışsınızdır. Can sıkıcı olabiliyor.

Discount servisimizin, aslında sistemde mevcut olan bir ürün için, “Böyle bir ürün yoktur.” şeklinde bir hata dönmesini istemeyiz. Açıkçası bu yöntemin en kritik noktası işte bu veri tutarlılığını sağlayabilmek. Bunun için bazı yöntemlerden bahsedeceğiz ancak bahsetmeden önce kişisel tavsiyem olarak şunu söyleyebilirim.

Eğer Discount servis kendi read-only Product tablosuna erişir ve ilgili veriyi bulamazsa, verinin gerçek sahibi olan Product servise anlık http isteği ile erişerek bir de oradan sorgulama yapabilir. Sorgu sonucu 2 ihtimallidir. Ürün yoksa, sorun da yok. Ancak ürün varsa, bu aradaki senkronizasyonun bozulduğu anlamına gelir. Bu durumda Discount service ürün bilgisine verinin ana kaynağından eriştiği için çalışmasına devam edebilir ve buradaki senkronizasyon bozukluğunu size bildirmek için bir event fırlatabilir.

Yani, “Ben x id’li ürünü kendi veri tabanımda bulamadım ama Product servise sorduğumda bana var olduğunu söyledi. Hayırdır?” anlamına gelen bir event’den bahsediyorum. Bu gibi event’leri dinleyen “Repair” rolünde ki farklı bir servis, bu x id’li ürünün Discount servisin Product tablosuna eklenmesini sağlayabilir. Tüm servisler bu beklenmedik durum için “Repair” servisine böyle bir event gönderebilirler.

Normal şartlarda, sistemde mevcut olmayan bir product id için discount servise bir indirim talebiyle gelinmesini beklemeyiz. Ancak aradaki senkronizasyonun bozulduğu durumlarda, önerdiğim yöntemle Discount servis çalışmasına devam edebilecek. Bu yöntemi uygulayarak servisleri birbirine bağımlı hale getirdiğimiz düşüncesine kapılmayın, çünkü Discount servis her işlemde yine ilk olarak kendi read-only tablolarına(product, customer) bakacak ve sadece çok nadir olmasını beklediğimiz senkronizasyon sorunlarında http isteği atarak ilerleyecek ve repair servisi haberdar edecek.

Kişisel önerimden bahsettiğime göre, veri tutarlılığını sağlayabilmek için uygulayabileceğimiz yöntemlerden bir kaçını çok detaya girmeden açıklayalım. Bu arada bu yöntemler, bu yazıda bahsettiğimiz veri paylaşımı metodunu olduğu kadar, tüm event-driven mimarileri ilgilendiren **data consistency** sorununun çözümü için geçerlidir.

- **Repair Service**

Yukarıda, hangi yöntemi uygularsanız uygulayın ekstra bir güvenlik önlemi olarak düşünebileceğiniz bir öneriden bahsederken değinmiş olduk aslında. Biraz daha açmak gerekirse, bütün işi read-only tabloların veri tutarlılığını sağlamak olan bir servis oluşturabiliriz. Bu servis her tetiklendiğinde esas veri kaynağı ile read-only tablolarının eşitlenmesi işini icra edecek. Bu eşitleme işlemini farklı yollarla yapılabilir.

- **Domain Event'lerin Persistent(kalıcı) olarak saklanması**

Event Sourcing yönteminde, event doğrudan bir bus yerine bir stream veya NoSQL veri tabanına yazılarak kalıcı olması sağlanır. Her listener servisi kendi veri tabanında bu event'lerin durumunu takip eder.

- **Outbox Pattern**

Bu yöntemde Domain Event'ler yine doğrudan bir bus'a yazılmıyor. Bunun yerine event'i fırlatan servisin kendi veri tabanında "outbox" rolündeki bir tabloya yazılıyor. Ancak burada kritik olan nokta, event'den önce yapılan işlemin ve outbox tablosuna yazılan event'in aynı transaction'ın bir parçası olması. Yani, sisteme yeni bir ürün eklendiğinde, ürün ekleme işlemi ve ProductCreated event'inin outbox tablosuna yazılması işlemi aynı transaction'da yapılarak event'in db'ye kaydedilmesi garantileniyor.

İkinci aşama ise, outbox tablosuna yazılan bu event'lerin bağımsız bir servis tarafından alınarak event bus'a yazılmasıdır. Bu bağımsız servis bu işlemi bir kaç farklı metotla yapabilir. Konuyla ilgili Chris Richardson'ın [burada](#) ve [burada](#) bahsettiği yöntemleri inceleyebilirsiniz.

- **Retry Policy**

Publisher servis tüm listener servislerden haberdardır ve her bir event'in ilgili listener'a başarılı bir şekilde iletildiğinden emin olur. Bu başarılı gönderim bilgisi "acknowledgement" olarak bilinir. Bu bilgiyi alana dek servisin event'i tekrar tekrar göndermesini

sağlayabilirsiniz. Bu şekilde en azından event'in listener tarafından alındığından ve işlendiğinden emin olunur. Burada kritik nokta, listener servisin "acknowledgement" bilgisini hangi aşamada gönderdiğidir. En doğru olanı, event'i alıp ilgili işlemi (Discount servisin yeni eklenen ürünü read-only tabloya kaydetmesi gibi) başarıyla yaptıktan sonra bu bilgiyi iletmesi olacaktır.

Bunun dışında farklı yöntemlerde mevcut, her bir yöntemin diğerlerine göre artıları ve eksileri olduğunu da belirtmekte fayda var.

Sonuç

Yazılım mimarilerinde bir sorunu çözmek veya bir kazanım elde etmek için yapılan her tercih yeni zorlukları da beraberinde getiriyor. Bu neredeyse her durumda geçerli bir kural adeta. Tıpkı, yazıda bahsettiğim, bir servisin başka bir servisin ihtiyaç duyduğu verisinin read-only bir kopyasını kendi veri tabanlarında tutması yönteminde olduğu gibi. Burada kazanımımız, tamamen izole ve otonom servisler elde etmek iken (ki büyük bir kazanım), bu read-only kopyanın esas veri kaynağı ile senkronizasyonu konusu ise yeni bir zorluğu beraberinde getiriyor.

Entegrasyon Testi

Integration Test yazmak hatırı sayılır bir efor gerektiriyor. **Unit Test** yazmaya kıyasla daha maliyetli ve açılması gereken kilit sayısı çok daha fazla. Unit Test ile sadece servisimizin fonksiyonelliğini garanti altına alırken, diğer servislerle olan iletişimimizi garantilemek için Integration Test'e ihtiyaç duyarız.

Konu entegrasyon olunca ve işin içine bir de Mikroservis Mimari girince zorluk seviyesi daha da artıyor tabi. Eğer daha önce **Monolith** bir uygulama için **Integration Test** yazdıysanız aradaki farkı daha net görebilirsiniz.

Bu bölümde **Mikroservis Mimari** üzerine kurulu sistemlerde entegrasyon testlerinin öneminden ve konuya farklı bir bakış açısı getirerek bizi Integration Test yazmanın maliyetinden kurtaran **Consumer Driven Contracts Testing** yaklaşımını inceleyeceğiz.

Entegrasyon Testi'ne Neden İhtiyaç Duyuyoruz?

Mikroservis Mimariler'de servis sayısı arttıkça, servisler arası iletişim ve entegrasyon sayısı da doğru orantılı olarak artacaktır. 10 adet servisten oluşan bir sistem ile 100 adet veya daha fazla servisin birbirleriyle konuştuğu bir sistemin karmaşıklığı ve bakım maliyetleri hiç şüphesiz aynı olmayacaktır.

Şimdi onlarca Mikroservis'den oluşan büyük ölçekli kurumsal bir uygulamayı ele alalım. **CustomerService** adında bir servisimiz olsun. Bu servis müşterilerimizle alakalı diğer servislerin ihtiyaç duyabileceği tüm servisleri sağlamakta. Dolayısıyla bu serviste yapılacak bir değişikliğin, servisi kullanan diğer servislere bildirilmesi gerekiyor. Aksi halde bu değişiklikten haberdar olmayan ilgili servis veya servisler, eğer şanslıysanız test ortamında hata verecektir. En kötü senaryoda ise, servis devam eden test süreçlerinin kapsamında değilse ancak canlı ortamda hata vermesiyle haberdar olacaksınız demektir ki bu en son isteyeceğimiz şey olur.

Aynı şekilde, eğer **CustomerService** de başka bir servisin verisine ihtiyaç duyuyorsa, o serviste yapılan değişikliğin de **CustomerService** tarafından bilinmesi gerekmektedir.

Peki bu değişiklik bildirimlerini nasıl yöneteceğiz? Bunu insan eliyle manuel olarak yönetebilir miyiz? Ayrıca, değiştirdiğimiz servisin hangi servisleri etkileyeceğini biliyor

muyuz? Bunun için sürekli güncel tuttuğumuz bir dokümantasyona ihtiyacımız var mı?

Eğer az sayıda Mikroservis'e sahip bir uygulamamız varsa, söz konusu ekipte küçük bir ekiptir muhtemelen ve bunu belki bir şekilde manüel ilerletebilirsiniz. Burada manüel'den kastımız ekip içerisinde, servislerde yapılan değişikliklerin, değişiklikten etkilenen servislerin sorumlularına doğrudan bildirilmesidir. Ancak onlarca hatta yüzlerce servisten bahsediyorsak bunu manüel olarak yönetmemiz söz konusu değil maalesef.

Çare: Consumer Driven Contracts Testing (CDC Testing)

Mikroservislerinizde yaptığınız en ufak bir değişikliğin servisi kullanan **Consumer**' ları nasıl etkilediğini bilmek gibi bir problemimiz var ve CDC testing bu problem için güzel bir çözüm olabilir.

CDC yaklaşımını en basit haliyle, iki servisin birbirlerine gönderdikleri verinin formatı konusunda anlaşmaya varması olarak tanımlayabiliriz. Bu yaklaşım, **Service Provider** veya **Service Consumer** tarafında yapılan her değişiklik bilgisinin anlık olarak paylaşılması prensibi üzerine kuruludur.

CDC'yi uygularken ki en önemli konulardan birisi Consumer ve Provider servislerini yöneten ekipler arasındaki iletişimdir. İletişim ne kadar kopuk olursa CDC'yi uygulamak da o denli zorlaşacaktır. Dolayısıyla en iyi senaryo, tüm servislerin aynı ekibin sorumluluğunda olduğu senaryodur.

Bu kadar konuştuk iyi güzel, peki bu **CDC**' yi nasıl uygulayacağız diye düşünmeye başladığınızı tahmin ederek size **Pact** framework'den bahsetmek isterim. Pact'in detaylı implementasyonu biraz uzun kaçabileceğinden burada sadece hangi yaramıza merhem olduğu, temel yapısı ve kullanımıyla ilgili giriş seviyesinde bilgi vermeyi uygun görüyorum.

Pact CDC Testing Framework

Pact'in official tanımına bakarak başlayalım;

Pact is a consumer-driven contract testing framework. Born out of a microservices boom, Pact was created to solve the problem of integration testing large, distributed systems.

Pact, **Pact Foundation** tarafından Ruby dili kullanılarak geliştirilen açık kaynak bir CDC Testing framework'üdür. Şuan Ruby'nin yanı sıra Php, Go, C# gibi birçok dil için desteği vardır. Pact Foundation'ın **github sayfasını** incellerseniz eğer her bir dil için ayrı bir repository üzerinden ilerlendiğini görebilirsiniz.

Şimdi çok teknik detayına ve kod kısmına girmeden adım adım implementasyonu nasıl yapacağımıza bakalım.

Senaryo

CustomerService adındaki bir servisimiz ve bu servisimizi kullanan **ProductService** isminde ikinci bir servisimiz olsun. Burada **CustomerService Provider**, **ProductService** ise **Consumer** rolündeler. Yani **CustomerService**' de yapılacak bir değişikliğin **ProductService**'i etkileyip etkilemediği konusu önem arz ediyor.

1- Consumer Servis Tarafında Contract Oluşturma

Consumer rolündeki **ProductService** servisi üzerinde eğer mevcut değilse bir test projesi oluşturuyoruz. Test yazmak istediğimiz senaryoları belirleyerek **Unit Test**'lerimizi **PactNet**'in belirlediği formata göre ve provider servisimizi **Mock**'layarak yazıyoruz. Burada **Unit Test** ifadesinin dikkatinizi çekmesi gerekiyordu. Baştan beri entegrasyon testi deyip duruyorduk nereden çıktı bu diye düşünebilirsiniz Şöyle ki;

CDC ile birlikte klasik bildiğimiz Integration Test yazmanın formatından çıkmış oluyoruz aslında. Bildiğiniz gibi normalde **Integration Test** metodları web service veya veri tabanı erişimlerini **Unit Test**'lerden farklı olarak **gerçekten** yapıyorlar. Yani bir **mocking** söz konusu değil. **CDC**'yi **Pact** ile implemente ederken de Unit Test yazar gibi, Provider servisleri mock'luyoruz çünkü bizim için artık önemli olan şey contracts yani Provider ile aramızdaki sözleşmemiz.

Consumer, **Provider**'a yapacağı **X**, isteğine karşılık **Y** tipinde bir yanıtı alması gerektiğini Unit Test sınıfları içerisinde **Pact**'in sağladığı fluent metodları kullanarak belirtir.

2- Pact Contract (Sözleşme) Oluşturulması ve Provider ile Paylaşılması

Consumer tarafında tüm test metotları yazıldıktan sonra testler çalıştırılır. Testler çalıştırıldıktan sonra json formatlı Contract'ımızın oluşması gerekiyor. Bu json dosyası

Pact'i configure ederken bizim belirlediğimiz bir dizinde oluşacaktır. Test metodlarında yapılan her değişiklikte bu json dosyası otomatik olarak güncellenecek ve belirlenen dizinde her zaman güncel hali yer alacaktır. **Yani sözleşmenin sürekli güncel tutulma işini Pact halletmekte ki bu çok önemli bir konu.**

Oluşan dosyanın json formatında ve oldukça okunabilir olması çok farklı yapıdaki servisler için bile **CDC** uygulayabilmemize olanak sağlıyor. **Go** dili ile yazılan bir **Provider** ile bir **Php Consumer**'ı Pact Contract'ı üzerinden el sıkışabiliyorlar.

Örnek bir Pact Contract'ı için aşağı yer almakta. **Interactions** listesi altında iki adet **interaction** olması, iki adet test metodu yazıldığı anlamına geliyor. **Request** kısmı **Consumer**'ın yapacağı istek, **Response** ise bu isteğe karşı **Consumer**'ın **Provider**'dan beklediği yanıtı ifade eder. Görüldüğü gibi aslında açıklamaya gerek bırakmayacak kadar okunaklı bir format. Kaynak : <https://github.com/tdshiple/pact-workshop-dotnet-core-v1>

```
1  {
2    "consumer": {
3      "name": "Consumer"
4    },
5    "provider": {
6      "name": "Provider"
7    },
8    "interactions": [
9      {
10       "description": "A invalid GET request for Date Validation with inval
11       "providerState": "There is data",
12       "request": {
13         "method": "get",
14         "path": "/api/provider",
15         "query": "validDateTime=lolz"
16       },
17       "response": {
18         "status": 400,
19         "headers": {
20           "Content-Type": "application/json; charset=utf-8"
21         },
22         "body": {
23           "message": "validDateTime is not a date or time"
24         }
25       }
26     },
27     {
28       "description": "A valid GET request for Date Validation",
29       "providerState": "There is no data",
```

```
30     "request": {
31         "method": "get",
32         "path": "/api/provider",
33         "query": "validDateTime=04/04/2018"
34     },
35     "response": {
36         "status": 404,
37         "headers": {
38         }
39     }
40 }
41 ],
42 "metadata": {
43     "pactSpecification": {
44         "version": "2.0.0"
45     }
46 }
47 }
```

3- Provider'ın Sözleşmeden Haberdar Edilmesi

Bu son aşamada artık Provider'ımıza kendisiyle anlaşma (contract) imzalamak isteyen **Consumer**'ları bildirip anlaşmaların nerede tutulduğu bilgisini vermemiz gerekiyor ki, **Provider**'ımız kendisinde yaptığı her değişiklikte hangi Consumer'ların etkilenip etkilenmeyeceğini bilsin.

Bu işlemi de yine Pact'in sağladığı **fluent** metodlar ile kolayca yapabiliyoruz. İstedığımız kadar Consumer ve Pact Url'ini Provider'ımıza tanımlayabiliyoruz. Kod detayına girmeden konuyu toparlamadan önce son olarak **Pact Broker** kavramından bahsedelim.

Pact Broker Nedir?

Oluşan json formatındaki contract'ımızı Provider ile paylaşma bölümünde '**Pact'i konfigure ederken bizim belirlediğimiz bir dizinde oluşacaktır.**' demiştik. **Consumer** ve **Provider**'ın aynı ortamda bulunduğu senaryolarda Contract'ı bir dizinde saklama işimizi görecektir ancak gerçek hayat senaryolarında bu dosyayı internet ortamında, yani bir sunucu üzerinden paylaşmanız gerekecek. **Pact** bize her iki imkanı da sağlamakta. Tek yapmamız gereken json dosyamızın konumu belirtirken dizin yerine **Pact Broker**'ımızın adresini ve varsa **credential** bilgilerini yazmak olacaktır. Aynı işlemi hem **Consumer** hem **Provider** tarafında yaptıktan sonra artık servislerimiz internet üzerinde **public** veya **private** olarak tutulan bir contract üzerinde el sıkışabilecekler.

Pact Broker hizmetini **SaaS** olarak almak isterseniz **official** bir hizmet de mevcut.

Ek olarak Pact Broker'ın arayüzünde Contract'ların son durumunu ve hangi servis hangi servislere bağımlı gibi verileri görebilmekte çok güzel. Örnek olarak Pact Broker'ın github sayfasından aldığım iki ekran görüntüsüyle konuyu sonlandırıyorum.

image from https://github.com/pact-foundation/pact_broker

Pacts







Consumer ↓↑		Provider ↓↑	Latest pact published	Last verified
Foo		Animals	2 minutes ago	2 days ago
Foo		Bar	7 days ago	15 days ago ⚠
Foo		Hello World App	1 day ago	
Foo		Wiffles	less than a minute ago	7 days ago
Some other app		A service	26 days ago	less than a minute ago
The Android App		The back end	less than a minute ago	

image from https://github.com/pact-foundation/pact_broker

Toparlarsak

Bu bölümde Mikroservis Mimari'lerdeki en zorlu konulardan birisi olan Integration Test konusuna farklı bir bakış getiren **Consumer Driven Contrats** yaklaşımını ve bu yaklaşımı uygulayabilmek için kullanılan **Pact** framework'ünü giriş seviyesinde inceledik.

Eğer ürünlerinizi Mikroservis Mimari ile geliştiriyorsanız, **Pact** veya muadili farklı bir framework'ü yazılım geliştirme yaşam döngünüzün olmazsa olmaz bir parçası haline getirmenizi tavsiye ederim.

Loglama ve Monitoring

Bu bölümde loglama için mimarimizi nasıl oluşturmamız, **best practice**'ler nelerdir sorularına yanıt arayacağız. Burada bahsedeceğim konuların bir çoğu sadece **Mikroservis** uygulamalara özel şeyler olmayacak, dolayısıyla **Monolith** mimaride uygulama geliştirenler için de ilgi çekici olacağını düşünüyorum. Konuyu iki ana başlıkta ele alacağız;

- Loglama Üzerine Genel Tavsiyeler
- Uçtan Uca Loglama Mimarisi

Loglama Üzerine Genel Tavsiyeler

Mikroservis Mimari'leri dağıtık sistemlerin bir türü olarak ele alabiliriz. Dağıtık sistemlerde her bir servisi ve bu servisin üzerinde çalıştığı sunucuyu **trace** etmek (izlemek), beklenmedik durumlarda hatanın kaynağına hızlı ve etkili şekilde inebilmek için çok önemlidir.

Servislerimizin her birisi farklı bir sunucuda sunulurken, oluşan uygulama loglarımız da haliyle dağıtık yapıda olacaktır. Buna ek olarak bir de **cloud** üzerinde **auto-scaled** bir yapınız varsa, o zaman hangi log hangi sunucuda ve ne zaman oluşmuş gibi soruların yanıtını almak için biraz daha fazla efor harcamanız gerekiyor. Zira bildiğiniz gibi **auto-scale** yapıda yük durumuna göre dinamik olarak yeni sunucular çok kısa sürede devreye alınıp aynı şekilde kapatılabiliyor. Loglama mimarimizi, bu kapatılan sunucularda log kaybı yaşamamak üzere kurgulamamız gerekmekte. Yani kısaca cluster'ınızın auto-scale olup olmaması da mimariyi kurgularken göz önünde bulundurmanız gereken konulardan birisi olmalı.

Logların Merkezileştirilmesi

Yapımız dağıtık bir yapı olduğundan, ilk düşünmemiz gereken konu tüm servislerimizin ürettiği logların tek yerden erişilebilir olmasıdır. Bir servisin logunu file'a, bir diğerinin DB'ye bir başkasının kuyruğa yazdığı bir ortamda oluşan bir hatanın kaynağına inmek saatler alabilir.

Merkezileştirme için çeşitli yöntemler mevcut. Mimari bölümünde biraz daha detaylı ele alacağımız için burada sadece kaçınmamız gereken bir **bad practice**'den bahsetmek isterim.

Mikroservis Mimari'yi uygularken, isteriz ki servislerimiz atomik olsun, yani sadece bir işe, bir domain'e odaklansın. Diğer servislere bağımlılığı sıfır noktasında olsun. Hal böyleyken her türlü ihtiyaç için irili ufaklı servisler oluşturmaya başlarız. İş loglamaya geldiğinde de yine bir http log service oluşturup loglama işini de bu servise yükleyerek diğer tüm servislerin bu servis üzerinden log atmasını isteyebiliriz. İlk bakışta her şey normalmiş gibi gözükse de servis sayısı, dolayısıyla üretilen log miktarı arttıkça loglama işlemi için oluşan http trafiği baş ağrısı olabiliyor. Yani bu maddeden çıkaracağımız ders, loglama işini üstlenen bir http service oluşturmaktan kaçınmamız gerektiği.

Merkezileştirme İçin Aggregation Tool Kullanma

Log Aggregation araçları, tüm loglarımızı tek bir ortak lokasyonda ve benzer formatta birleştirme ihtiyacı üzerine ortaya çıkmıştır. Bu birleştirme işi için farklı yöntemler mevcut. Burada seçeceğimiz yöntem, servis loglarınızı nerede ve nasıl tuttuğunuza göre belirlenecektir.

Örneğin, siz her servisin kendi sunucusunda bir dizine .txt olarak log atmasına karar verdiniz. Bu logları birleştirme için bir job yazar, belirli aralıklarla tek ortak bir dizine tüm txt'leri toplayabilirsiniz. Tercih kötü olsa bile sonuçta bir **log aggregation** yapmış olursunuz aslında.

Daha doğru olan ise, bu iş için geliştirilen ücretli veya ücretsiz bir araç kullanmanız yönünde. **ELK (elastic search, logstash,kibana)** stack'inin aggregation aracı olan **logstash**'i önerebilirim mesela. ELK'dan, mimari bölümde bahsedeceğimiz için burada fazla detaya girmiyorum

Log'daki Tüm Alanların Aranabilir Olması

Loglarınızı sorgularken bazı alanlar üzerinden daha sık sorgulama yapmanız gayet doğaldır. Ancak siz her durumda mevcut alanların tümü üzerinden sorgulama yapabilecek ve hızlıca yanıt alabilecek bir yapı oluşturmaya odaklanırsanız iyi edersiniz.

Hız için veri tabanı üzerinde index oluşturma tabi ilk aklımıza gelen çözüm. **CustomerId**, **UserId**, **LogLevel** gibi alanlar üzerinden index'lenme olmazsa olmazdır, ancak kritik bir anda örneğin **InstanceId** alanı üzerinden sorgulama yapmanız gerekirse, dakikalarca beklemek biraz can sıkıcı olabilir.

Eğer çözüm index oluşturmaksa neden tüm alanlar için oluşturmuyoruz diye düşünebilirsiniz. Index'leme işi maliyetlidir. Çok fazla veya çok büyük boyutlu index'ler memory'de büyüklüğüne göre yer kaplayacaktır. Üstelik bununla da kalmaz, her bir insert/delete/update komutundan sonra bu indexlerin güncellenmesi gerekmektedir. Bu da sizin insert/delete/update sürelerinizin uzamasına yol açabilir.

Ancak bu index'leme işini uygulama veri tabanında değil de ,bir log aggregation aracı üzerinde yaparsanız performans kaybının önüne geçebilirsiniz.

Log Seviyesinin (Log Level) Dinamik Ayarlanması

Atılan her bir satır log'un mevcut kaynaklarınızı tükettiğini ve aynı zamanda log sorgulama maliyetinizi (süresini) artırdığını unutmamanız gerekiyor. Bu bağlamda, servislerimiz ayaktaiken, **runtime**'da log seviyemizi dinamik olarak,uygulamamızı kesintiye uğratmadan değiştirmek isteyebiliriz.Elbette kullandığımız yardımcı kütüphanenin (varsa) bunu destekliyor olması gerekmekte.

Bu özelliği ağırlıklı olarak **Production** ortamımızda kullanabiliriz. Örneğin geçici bir süreliğine Debug seviyesinde log atılmasını isteyebiliriz. Servisimiz ayağı kalkarken en detaylı log seviyesini pasife çekip, sonrasında aktif hale getirerek aslında bize bir faydası dokunmayan, gereksiz bir sürü logdan kurtulabiliriz.

Asenkron Loglama

Loglama işlemini hangi yolla yaparsanız yapın, log'u atacak olan birimin asenkron çalışması uygulama performansı açısından önemlidir.

Asenkron loglamada, loglamayı yapacak olan **logger thread**, o an yürütülmekte olan transaction'ın (http isteğinin) thread'ini block'lamayacağı için herhangi bir performans kaybına neden olmayacaktır. Ek olarak uygulamanızın loglama işleminin sonucuyla da ilgilenmemesi gerekiyor. Bir başka deyişle loglar **fire-and-forget** prensibi ile gönderilmelidir.

CorrelationId Kullanımı

Bir transaction'ın icra edilmesi için arka planda 3–5 farklı web servisin birbirlerini tetiklediği bir senaryoyu ele alalım. Burada yapılan işlem aslında tek bir işlem. Örneğin bir satın alma işlemi. Ancak arka planda 5 farklı servis çalışmakta ve, 5 adet log atıldığını biliyoruz. Bu işlemin loglarına erişmek istediğimizde bu 5 adet log için bir grup numarası olsaydı ve bu no ile sorgulama yaptığımızda sadece bu 5 adet logu görebilseydik güzel olurdu değil mi?

Bu gruplamayı yapmak için **X-Correlation-Id** http header'ını kullanabiliriz. Bu header'a işlem bazında **unique** bir değer set etmemiz gerekmekte. Ek olarak log tablomuza da CorrelationId adında bir alanda eklememiz gerekiyor tabi. Artık geriye, bu header'ı her servis **request** ve **response**'uiçin doğru şekilde set etmek olacaktır. Örnek senaryomuzdaki 5 adet servisten her birisi bir sonraki servise bu CorrelationId değerini geçecektir. Kulaktan kulağa oyunu gibi düşünebilirsiniz.

Detaylı ve Anlaşılabilir Log

Gün içinde bir servisin loglarını inceleme gereği duyuyorsak, bir şeyler ters gidiyor demektir, tabi istisnai durumlar olabilir. Gerek development aşamasında gerekse production ortamımızda loglar tabiri caizse elimiz ayağımız oluyor. Tabi yeteri kadar detaylı bilgi verebildikleri zaman. Bu da yine bizim elimizde olan bir şey. Özellikle **error log**'ları mümkün olduğunca detaylı ve önemli bilgi içermelidir. Detaylı yanında önemli de dedim çünkü gereksiz bir sürü detay içeren ama hatanın kaynağına inmeye yardımcı olacak bilgiyi içermeyen logun kimseye bir faydası olmayacaktır.

Sözün özü, eğer loglarınızı incelerken “Keşke şu bilgi de elimizde olsaydı” diye iç geçiriyorsanız, o bilgi için de ek bir alan açmanın zamanı gelmiş demektir.

Log Zamanı İçin Utc Time Kullanımı

Özellikle servislerin cloud üzerinde sunulduğu senaryolarda, mevcut sunucuların veya yük oluşması anında devreye alınacak yeni sunucuların hangi time zone'da olacağıyla ilgilenmek istemezsiniz. Sizin için önemli olan sunucunun performansı ve yüksek erişilebilirliği olacaktır.

Ancak iş loglarınızı sorgulamaya gelince, log zamanının hangi time zone'a göre atılacağı konusu önem arz ediyor. Zamana göre order by ettiğiniz logların işlem sırasına göre hatalı olarak gelmesi istemezsiniz. Burada tavsiyem Utc time kullanmanızdır. Böylece çok farklı lokasyonlardaki servislerden atılan loglarda bile zaman karmaşası yaşamamış olursunuz. Bu arada eğer sizin için logu atan sunucunun local zamanı da önemliyse, log veri tabanınıza **TimeZone** adında opsiyonel bir alan ekleyip time zone bilgisini de buraya yazabilirsiniz.

Instance Id Ekleme

Eğer servislerinizi herhangi bir cloud provider üzerinden sunuyorsanız ve auto-scaling özelliğini kullanıyorsanız, yukarıda da bahsettiğim gibi yük durumuna göre sürekli yeni sunucular (instance) eklenip çıkarılacaktır. Log'un hangi instance'dan atıldığı, logu atan sunucunun hala aktif olup olmadığı gibi konular sizin için önemliyse, logunuza **InstanceId** alanını ekleyerek bu sorunu çözebilirsiniz.

Log Alert Sistemi

Loglama alt yapınız artık olgunlaştı, loglarınız sorunsuz şekilde akıyor, sorgular gayet hızlı çalışıyor ne mutlu size. Artık işi bir adım öteye taşımak gerekiyor.

Eğer canlı ortamda oluşan hatalarınız için biraz daha proaktif olmak ve daha hızlı aksiyon almak istiyorsanız mutlaka bir alarm sisteminin devreye almanız gerekiyor. Bu sistemi sağlayan ücretli ücretsiz birçok **third party** araç mevcut olmakla birlikte kendi implementasyonunuzu da pek ala yapabilirsiniz tabi.

Bu tarz alarm yapılarıyla örneğin, gelen log daki **LogDetail** alanında "**DB Connection Timeout**" hatası geçiyorsa şu kişi veya kişilere slack'ten mesaj at veya sms at gibi aksiyonlar alabildiğimiz için oldukça faydalı olduğunu söyleyebilirim.

Alarm konusu için bir süre deneyimleme fırsatı bulduğum **Seq** i incelemenizi öneririm. Tabi **ELK** ile de veri değişikliklerini izleyerek aynı şekilde alarm kuralları oluşturabilmeniz mümkün.

Uçtan Uca Loglama Mimarisi

Bu bölümde, yukarıda yaptığımız öneriler ışığında örnek bir mimari tasarlayıp, çizdiğim diagram üzerinden yorumlamaya çalışacağım.

Hatırlarsanız loglamayı bir servis üzerinden http request ile yapmanın kötü bir fikir olduğundan bahsetmiştik. Ancak aynı zamanda logları bir şekilde merkezileştirmemiz gerektiğini de söyledik. Yapmamız gereken şey çokta karmaşık değil aslında.

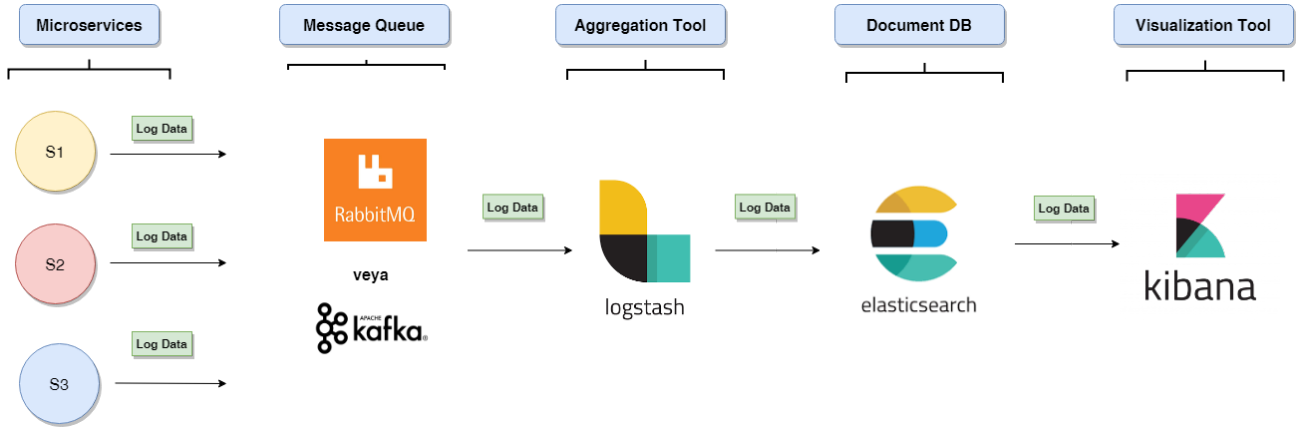
Her Mikroservis, oluşturduğu logları aracı bir başka http servis olmaksızın asenkron olarak bir kuyruğa gönderebilir. Buradaki kuyruk, **Apache Kafka** veya **RabbitMQ** gibi bir **message broker** olabileceği gibi, kuyruk kullanmadan doğrudan bir **RDBMS** veya **NoSQL** veri tabanı da olabilir. Bu tamamen sizin log yoğunluğuna bağlı olarak yaptığınız tasarıma kalmış. Benim tavsiyem, **Kafka** veya **RabbitMQ** ikilisinden birini kullanmanız yönünde olur. Eğer doğrudan bir veri tabanına yazacaksanız da bu RDBMS yerine bir NoSQL veri tabanı olmalı.

Loglarınızı doğrudan bir kuyruğa göndererek veri tabanı sunucunuzu ekstra ve büyük bir yükten kurtarmış olursunuz. Veri tabanı sunucunuz sizin en değerli kaynağınızdır. ve unutmayın DB sunucuları ram'i sever yani bol bol kullanır.

Bir diğer aklıma gelen konu da uygulamanız ayağı kalkarken, henüz veri tabanı bağlantısı yapılmadan önce loglamaya değer bazı bilgileriniz olabilir ancak veri tabanı bağlantısı henüz sağlanmadığı için bu bilgileri loglayamayacaksınız.

Kuyruk yapıları AMQP protokolünü desteklerler. Bu protokol Http'nin aksine asenkron çalışır ve mesajları alıcısına teslim etme garantisi verir. Tabi sizin kuyruğu nasıl konfigüre ettiğiniz de önemlidir. Servisler arası iletişimi incelediğimiz bölümde AMQP'den daha detaylı bahsetmiştik hatırlarsanız.

Tüm bu bilgiler ışığında çizdiğim aşağıdaki basit mimariyi siz de uygulamalarınız için rahatlıkla uygulayabilirsiniz.



Mikroservis Uygulamalar İçin Loglama Mimarisi

Son olarak mimariyi oluşturan bu 5 parçadan kısaca bahsetmek gerekirse;

Microservices

Farklı format ve yoğunluklarda loglar üreten ve kuyruğa gönderen mevcut microservislerimiz mimarinin ilk bacağını oluşturmakta. Dikkat ederseniz servisler ve kuyruk arasında başka aracı bir servis bulunmuyor.

Message Queue

Mikroservis'lerimizin ürettikleri logları **publish** ettikleri kuyruğu tanımladığımız message broker'ımız. Log yoğunluğuna göre bu broker'ları da yatayda ölçekleyebilirsiniz yalnız, Kafka yatayda ölçeklenebiliyorken yanılmıyorsa RabbitMQ bu imkanı vermiyor. RabbitMQ da, iki farklı sunucu kurup aktif-aktif çalışmasını sağlayabiliyorsunuz ancak yükü 2 sunucu arasında dağıtarak yatayda ölçekleneyim diyemiyorsunuz.

Aggregation Tool

Aslında ELK bütün olarak bir log aggregation aracıdır demek de yanlış olmaz. **Logstash**, logları ilk karşılayan, isteğinize göre manipüle edebilen ve sonrasında Elasticsearch'e gönderen taraf olduğu için ben, Logstash ELK stack'inin log aggregation tool'udur demeyi tercih ediyorum.

NoSQL Document DB

Elasticsearch, bir arama motoru gibi alıřmak zere optimize edilmiř, NoSQL document tipi veri tabanıdır. ELK stack'inde logların fiziksel olarak saklandıęı, indexlendięi ve sorgulandıęı yer burasıdır.

Visualization Tool

ELK stack'ini tercih ettięimiz iin loglarımızı listelemek, sorgulamak ve eřitli faydalı grafiklerden faydalanmak iin bu stack'in sunmuř olduęu Kibana visualiztion tool'unu kullanıyoruz. Kibana, kullanıřlı arayz, log sorgularken iřimizi kolaylařtıran query syntax'ı, oluřturulan log raporlarını excel export edebilme gibi birok zellięi bnyesinde barındırıyor.

Sonuç

Mikroservis Mimari'nin, bir bütünü küçük parçalara ayırarak yönetme fikrini akla getirmesinden ötürü cezbedici bir tarafının olduğu kesin. Bu cezbedicilik ve büyük firmaların ardarda yaptığı Mikroservis Mimari dönüşümleri **hype** rüzgarını da beraberinde getirdi. Bunun sonuncunda da bu dönüşümü başarıyla tamamlayanların yanında hatırı sayılır bir miktarda da başarısızlık hikayesi ortaya çıkmış oldu.

Küçük veya orta büyüklükte, henüz canlıya bile çıkmamış , hatta belki ölçeklenme ihtiyacı bile olmayan uygulamalarımızı Mikroservis Mimari'ye dönüştürmek için kolları sıvadık ve gün sonunda aynı ortak veri tabanını kullanan, birbirleriyle http isteği yaparak haberleşen, dolayısıyla bir birlerine göbekten bağlı olan web servislerimiz oldu.

Kişisel bloğumdaki yazı dizisine ve bu e-book'u hazırlamaya başlarken ki motivasyonum, az da olsa biraz farkındalık oluşturabilmek ve henüz işe koyulmamış, araştırma safhasındaki ekiplerin aklına 'acaba' sorusunu getirebilmektir.

Acaba ihtiyacınız olan şey tam olarak bu mu, yoksa hype rüzgarına mı kapıldınız?

Başta medium olmak üzere, twitter ve linkedin üzerinden soru ve eleştiriler göndererek yazılarak katkıda bulunan herkese çok teşekkür ederim. Her türlü öneri ve düzeltme için [buradan](#) issue oluşturabilir veya [twitter](#) üzerinden iletişime geçebilirsiniz.