1-What is the difference between manual testing and automated testing ?

Manuel test ile otomasyon testi arasında bazı büyük farklar vardır. Manuel testte, bir insan testleri test komut dosyaları olmadan adım adım gerçekleştirir. Otomatik testlerde testler, diğer araçlar ve yazılımlarla birlikte test otomasyon çerçeveleri aracılığıyla otomatik olarak yürütülür.

2-What does Assert class ?

Assert, bir test senaryosunun Başarılı veya Başarısız durumunu belirlemede yararlı bir yöntemdir. Assert yöntemleri, Java.lang.Object sınıfını **genişleten** org.junit.Assert sınıfı tarafından sağlanır.

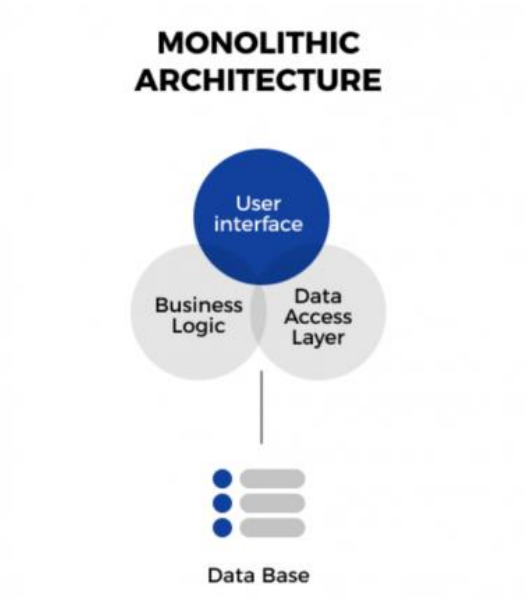
Boolean, Null, Identical vb. gibi çeşitli iddia türleri vardır.

Junit, test senaryoları yazmak ve test başarısızlığını tespit etmek için yararlı olan bir dizi onaylama yöntemi sağlar

3 - How can be tested 'private' methods ?

- Metotlara kendi package yapısında erişime izin verilir.(fakat encapsulation ' a zarar verebilir)
- Static olarak sınıfın içine bir test metodu yerleştirilir(karışıklık yaratacağından çok tercih edilmeyebilir)
- Yansıtma yöntemi objeler dışarı alınarak taklit edilir yansıtma nın çıktıları bize private ın çıktıların verir.
- TDD=Test Driven Design ' ın doğru bir şekilde işletilmesiyle private methodun test edilmesi gerekliliği ortadan kalkar.

4-What is Monolithic Architecture ?



İngilizce olarak Monolithic Architecture olarak geçen bu mimari yaklaşım, uygulamanın tüm parçalarının tek bir çatı altında geliştirilmesi ve sunulması olarak tanımlayabiliriz. Birçoğumuz sektöre girdiğimizde adını bilmediğimiz bu mimari yaklaşım ile devam ederiz zaten.

Üzerinde çalıştığımız/çalışacağımız projelerde, yeterli insan kaynağı olmadığından veya zaman yetersizliğinden dolayı doğru analiz ve projelendirme yapamaz ve hızlı olarak productiona geçmek için monolith mimariyle devam ederiz.

Fakat zamanla ekip büyütme ihtiyacı ve optimizasyon içinden çıkılmaz zorlu ve yorucu bir hal alır. Bu durumlarda yeni arayışlara girer, görev paylaşımı yaparız. Açık konuşmak gerekirse, seed yatırımı almamış çoğu girişim bu süreci tam olarak böyle yaşar. Bir taraftan müşteri taleplerini karşılamaya çalışırken diğer taraftan projenin kontrolden çıkmasını yönetmek zorunda kalırsınız. İşte bu noktada yeni arayışlara geçerek eninde sonunda Mikroservis mimarisinde karar kılarsınız.

Ama Mikroservise geçmeden önce gelin Monolith yapının avantajları ve dezavantajlarını inceleyelim:

Avantajlar

- Küçük ekipler için yönetilmesi ve geliştirmesi kolaydır.
- CI/CD ve Test süreçleri neredeyse DevOPS gerektirmez.
- Birden fazla ekip ihtiyacı duymaz aynı anda birden fazla modülle çalışılabilirsiniz (Bunun bir avantajdan ziyade dezavantaj olduğunu göreceğiz)

Dezavantajlar

  - Zamanla yeni feature ekleme ve optimizasyon zorlaşır.
  - Bağımlılık ve Kod tekrarından dolayı versiyonlar arasında sorunlar yaşanır.
- Proje bir bütün olduğundan, projenin bir bölümü auto scale yapılamaz tüm projeyi scale etmeniz gerekir.
- Farklı framework/programlama dili kullanarak, daha stabil ve uygun maliyetle sunabileceğiniz modülleri proje içinde yazmanız gerekir (örneğin: AWS Lambda kullanmak)
- Tüm proje aynı veritabanını kullanacağından performans optimizasyonu gerekir.
- Sık kullanılan ve yüksek kaynak kullanımına sahip bir modülünüz için tüm sistem kaynaklarını arttırmanız gerekir.
- Projeye yeni dahil olacak geliştiriciler için anlaması güç ve zaman alacak bir code base sunar.

5-What are the best practices to write a Unit Test Case ?

1-Okunabilir Tesler Yaz

Okunması kolay testler, kodunuzun nasıl çalıştığını, amacını ve test başarısız olduğunda neyin yanlış gittiğini anlamak için rahatlatıcıdır. Kurulum mantığını ilk bakışta ortaya koyan testler, kodu hata ayıklamadan sorunun nasıl çözüleceğini bulmak için daha uygundur. Bu tür bir okunabilirlik, üretim kodu değişikliklerinin de testlerde güncellenmesi gerektiğinden, testlerin sürdürülebilirliğini de geliştirir.

- İlk olarak, her test durumu için sağlam bir adlandırma kuralına sahip olun. Testleri, konuyu, hangi senaryonun test edildiğini ve beklenen sonucu anında açıklayacak şekilde adlandırın.

- İkinci olarak, test aşamalarını net bir şekilde tanımlamak ve okunabilirliği artırmak için Düzenle, Harekete Geç, Assert Et modelini kullanın.

2. Magic number ve magic String 'lerden kaçının

Magic Stringlerin veya sayıların kullanılması, testleri daha az okunabilir hale getirdiği için okuyucuların kafasını karıştırır. Ek olarak, okuyucuları uygulama ayrıntılarına bakmaktan alıkoyar ve gerçek teste odaklanmak yerine neden belirli bir değerin seçildiğini merak etmelerini sağlar. Sihirli sayıya sahip örnek bir kod parçası:

Magic number ile test

Öte yandan, bir sabitin değiştirilmesi gerekiyorsa, onu bir yerde değiştirmek diğer tüm değerleri günceller. Bu nedenle, testlerde değer atamak için değişkenleri veya sabitleri kullanmak daha iyidir. Testler yazarken mümkün olduğunca çok niyet ifade etmenize yardımcı olur. Şimdi Magic number'ı okunabilir adı olan ve sayının anlamını açıklayan bir sabitle değiştirelim.

3. Deterministik Testler Yazın

Deterministik testler ya her zaman geçer ya da sabitlenene kadar her zaman başarısız olur. Ancak kod değiştirilmedikçe her çalıştırıldıklarında aynı davranışı sergilerler. Bu nedenle, bazen geçen ve bazen başarısız olan deterministik olmayan bir test yapmaktansa, hiç test yapmamak kadar iyidir.

4-Test bağımlılıklarından kaçının

Test çalıştırıcıları genellikle belirli bir sıraya bağlı kalmadan aynı anda birden fazla birim testi çalıştırır, bu nedenle testler arasındaki karşılıklı bağımlılıklar onları kararsız hale getirir ve yürütmeyi ve hata ayıklamayı zorlaştırır. Test karşılıklı bağımlılıklarından kaçınmak için her test senaryosunun kendi kurulum ve sökme mekanizmasına sahip olduğundan emin olmalısınız.

Örneğin, test çalıştırıcısının bir süre belirli bir sırayla birkaç test çalıştırdığını ve kendi kurulumu olmadan yeni bir testin eklendiğini varsayalım. Şimdi, test çalıştırıcı yürütme süresini azaltmak için tüm testleri paralel olarak çalıştırırsa, tüm test takımının yönünü şaşırtır ve testleriniz başarısız olmaya başlar.

Şimdi soru, tamamen bağımsız testler nasıl yazılır?Birincisi, test senaryolarını yazdığınız sıraya göre hiçbir şey varsaymamak. Testler birleştirilirse, kodu bağımsız olarak test edilecek küçük gruplara/sınıflara ayırın. Aksi takdirde, bir birimdeki değişiklikler diğer birimleri etkileyebilir ve tüm paketin başarısız olmasına neden olabilir.

5-Mantıksal koşullar ve manuel dizelerle birim testleri yazma

birleştirme, test takımınızdaki hata olasılığını artırır. Testler, uygulama ayrıntıları yerine beklenen sonuca odaklanmalıdır. if, while, switch, for, vb. gibi koşulların eklenmesi, testleri daha az belirleyici ve okunabilir hale getirebilir. Bir teste mantık eklemek kaçınılmaz görünüyorsa, testi iki veya daha fazla farklı teste bölebilirsiniz. Örneğin, aşağıdaki koda (with logic) bir göz atın.

```
@Test
public void testsTotalPriceAsSumOfProductPrices() {
    Products products = new Products(); // Or any appropriate constructor
    products.add(new Product("first", 10)); // Assuming such a constructor exists
    products.add(new Product("second", 20));
    products.add(new Product("second", 30));
    assertEquals("Sum must be eq to 60", 60, products.getTotalPrice());
}
```

SIMFORM

Şimdi, aşağıdaki yeniden düzenlenmiş koda (without logic) bakın. mi?

```
@Test
public void testsTotalPriceAsSumOfProductPrices() {
    Products products = new Products();

    addProductsWithPrices(products, 10,20,30);

    assertEquals(60, products.getTotalPrice());

}

private static void addProductsWithPrices(Products products, Double...prices){
  for(Double price : prices){
     //could keep static counter or something
     //to make names unique
     products.add(new Product("name", price));
  }
}
```

SIMFORM

6-Tek bir birim testinde birden çok Assert kaçının

Bir birim testinin etkili olması için, her seferinde bir kullanım senaryosu tutun, yani testlerde yalnızca bir iddiaya sahip olun. Tek bir teste birden fazla onaylama eklerseniz ne olacağını merak ediyorsanız, basit bir örnek verelim.

Bazen daha fazla özelliği kapsamak için bir sete 10'dan fazla onaylama dahil edilir. Bu tür durumlar, tek bir arıza meydana gelmiş olsa bile, sorunun temel nedenini kontrol etmek için tüm iddiaların üzerinden geçilmesiyle sonuçlanır. Ayrıca, bir iddianın başarısız olması durumunda diğer iddialar asla kontrol edilmez ve bu da testin başarısız olduğuna dair net olmayan bir vizyona neden olur.

Her Assert için ayrı test komut dosyaları yazmak sıkıcı görünse de, genel olarak daha fazla zaman ve çaba tasarrufu sağlar ve uzun vadede daha kesindir. Aynı testi farklı değerlerle birden çok kez çalıştırmanıza izin verdiği için parametreli testleri de kullanabilirsiniz.

7-Testlerinizi çok fazla implemantasyon detayından uzak tutun

implemantasyon kodunda yapılan en küçük değişiklikler için bile başarısız olmaya devam ederse, testlerin sürdürülmesi zordur. Bu nedenle en iyi seçenek, implemantasyon ayrıntılarını uzak tutmak ve testleri tekrar tekrar yazmaktan zaman kazanmaktır. Bu nedenle, implemantasyon detayları ile birleştirme testleri, testlerin değerini düşürür.

Birim testleri, üretim kodunun dahili özellikleriyle yoğun bir şekilde eşleştirilmedikleri takdirde değişime karşı daha dirençlidir. Ayrıca geliştiricilerin gerektiğinde yeniden düzenleme yapmasına olanak tanır ve bir güvenlik ağı ile değerli geri bildirim sağlar.

8-Testleri geliştirme sırasında yazın,

Birim testleri, test piramidinin tabanında yer alır ve geliştirme döngüsünde gerçekleştirilen en eski testlerdir. Bu nedenle, geliştirmeden sonra değil, geliştirmenin yanında çalıştırıldıklarında en iyi şekilde çalışırlar.

Birim testlerini olabildiğince erken ayarlamak, temiz kod yazmayı ve hataları erkenden tanımlamayı destekler. Geliştirmenin sonunda testler yazmak, test edilemeyen kodlara neden olabilir. Aksine, üretim koduna paralel testler yazmak, hem test kodunu hem de üretim kodunu birlikte incelememizi sağlar. Ayrıca geliştiricilerin kodu daha iyi anlamasına yardımcı olur. Ayrıca birim test sürecini daha ölçeklenebilir ve sürdürülebilir hale getirir.

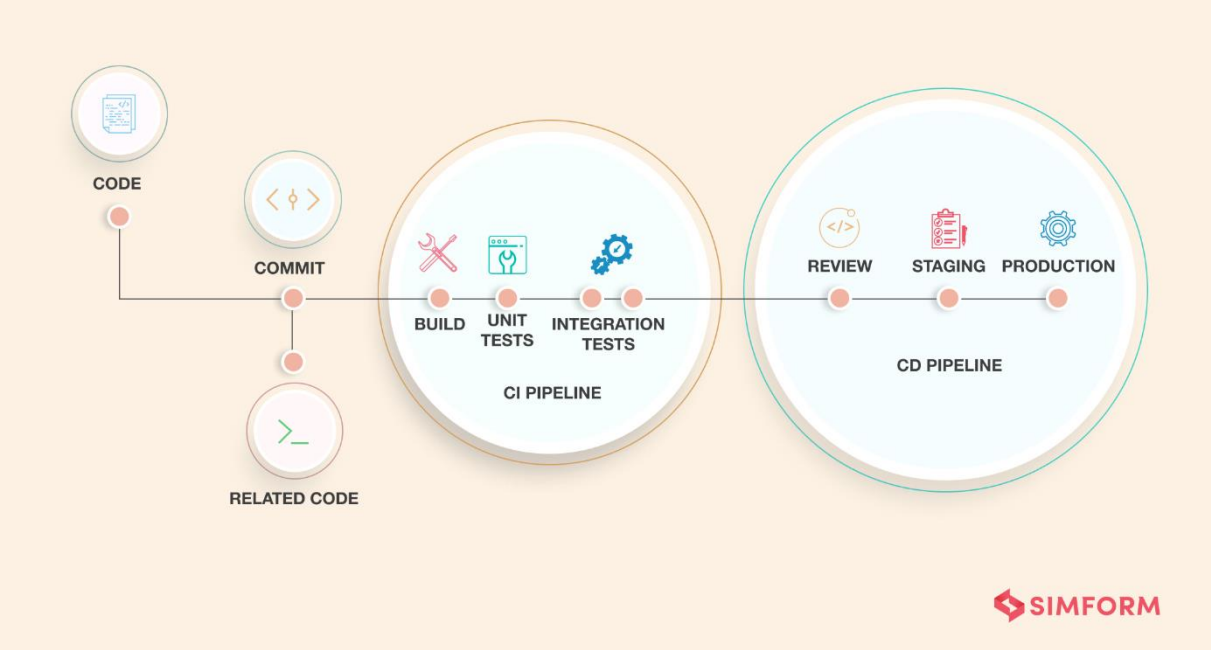
Bazı kuruluşlar, esasen Test odaklı geliştirme (TDD) haline gelen üretim kodundan önce birim testleri yazmayı tercih eder. Bu uygulama, kodun beklenen davranışına zihinsel olarak hazırlanmanıza yardımcı olur. Geliştirme sürecinde sormanın aksine, siz kodu yazmaya başlamadan önce bile sorular ve vakalar ortaya çıkarır. TDD yaklaşımı, yeniden işleme ve hata ayıklama için harcanan süreyi azaltmaya yardımcı olur. Son yeniden düzenlemenin daha önce çalışan kodu bozup bozmadığını anında size söyler. Birim testiyle birlikte hataları ve sorunları hızlı bir şekilde tanımlayabilir

9-CI/CD araçlarını kullanarak testleri otomatikleştirin

Testleri bir CI/CD ardışık düzenine dahil ederek otomatikleştirmek, testleri günde birden çok kez kolayca çalıştırmanıza olanak tanır. Her kod taahhüdünde sürekli test ve test yürütme sağlar. Bir test çalıştırmayı unutsanız bile, CI (Sürekli Entegrasyon) sunucusu, müşterilere buggy kodunun iletilmesini engeller .

Aksine, manuel testler yeterli sayıda testi hızlı, uygun ve doğru bir şekilde çalıştıramaz. Yazılımı kullanıma sunmak için son teslim tarihlerinin kısıtlı olması özellikle zorlaşıyor.

Otomatik testler, hataların erken tespit edilmesine yardımcı olur, hızlı geri bildirim sağlar ve ekstra bir güvenlik katmanı ekler. Ayrıca, derinlemesine analiz ve geliştiricilerin verimli çalışmasına olanak tanıyan kod kapsamı, değiştirilmiş kod kapsamı, kaç testin çalıştığı, performans vb. hakkında bilgiler sağlar.



10-Testleri periyodik olarak güncelleyin

Birim testleri, kodu ve davranışını daha kolay anlaşılır hale getirerek ayrıntılı belgelerle yeni ekip üyelerine yardımcı olduğu için uzun vadeli projeler için idealdir. Bu nedenle, testleri periyodik olarak sürdürmek ve güncellemek, onları yararlı belgeler oluşturmak için ideal test takımları haline getirir. Bu kaliteye sahip olmayan birim testleri, sonunda ekibinizin iş ilerlemesini yavaşlattığı için daha az kullanışlıdır.

6 - Why does JUnit only report the first failure in a single test ?

Tek bir testte birden fazla hatanın rapor edilmesi, genellikle testin çok fazla şey yaptığının ve bir birim testinin çok büyük olduğunun bir işaretidir. JUnit, bir dizi küçük testle en iyi şekilde çalışacak şekilde tasarlanmıştır. Her testi, test sınıfının ayrı bir örneği içinde yürütür. Her testte başarısızlığı bildirir.

7-What is the role of actuator in spring boot ?

Esasen, Aktüatör, uygulamamıza üretime hazır özellikler getiriyor.

Uygulamamızı izlemek, ölçümleri toplamak, trafiği anlamak veya veritabanımızın durumu bu bağımlılıkla önemsiz hale gelir.

Bu kitaplığın ana yararı, bu özellikleri kendimiz uygulamak zorunda kalmadan üretim düzeyinde araçlar elde edebilmemizdir.

Aktüatör, esas olarak, çalışan uygulama hakkında - sağlık, metrikler, bilgi, döküm, env vb. - operasyonel bilgileri ortaya çıkarmak için kullanılır. Bizim onunla etkileşim kurmamızı sağlamak için HTTP uç noktalarını veya JMX çekirdeklerini kullanır.

Bu bağımlılık sınıf yolunda olduğunda, kutunun dışında bizim için birkaç uç nokta mevcuttur. Çoğu Spring modülünde olduğu gibi, onu birçok şekilde kolayca yapılandırabilir veya genişletebiliriz.

8- What are the benefits and drawbacks of Microservices ?

Mikroservice Avantajları

Mikroservis bağımsız, bağımsız dağıtım modülüdür.

Ölçeklendirme maliyeti, monolitik mimariye göre nispeten daha düşüktür.

Mikroservisler" bağımsız olarak yönetilebilen servislerdir. İhtiyaç oldukça daha fazla servise olanak sağlar. Mevcut servis üzerindeki etkiyi en aza indirir.

Uygulamanın tamamında yükseltme yapmak yerine her bir servisi ayrı ayrı değiştirmek veya yükseltmek mümkündür.

Mikroservisler, son zamanlarda daha fazla işlev veya modül ekleyerek yükseltilen bir uygulama geliştirmemize izin verir.

Olay akışı teknolojisinin, ağır araya giren iletişimle karşılaştırıldığında kolay entegrasyona olanak tanımasını sağlar.

Mikroservisler, tek sorumluluk ilkesini takip eder.

Performansı artırmak için talep edilen servis birden fazla sunucuya dağıtılabilir.

Daha az bağımlılık ve test edilmesi kolay.

Dinamik ölçekleme.

Daha hızlı serbest bırakma döngüsü.

MikroServis Dezavantajları

Mikroservisler, dağıtılmış sistemin tüm ilgili karmaşıklıklarına sahiptir.

Farklı servisler arasındaki iletişim sırasında daha yüksek bir hata olasılığı vardır.

Çok sayıda servisi yönetmek zordur.

Geliştiricinin ağ gecikmesi ve yük dengeleme gibi sorunu çözmesi gerekiyor.

Dağıtılmış bir ortam üzerinde karmaşık testler.

9-What are the challenges that one has to face while using Microservices ?

Tek bir mikro servis projesi genellikle iyi çalışabilir. Teknik zorluklar da olsa, zorlukların çoğu, birden fazla mikro servis koordine edilmesi gerektiğinde aniden ortaya çıkan klasik iletişim sorunlarından kaynaklanmaktadır.

Eski tarz monolitlerin aksine ayrıştırılmış, işlem yapmayan sistemler tasarlamak zordur.

Geleneksel veritabanlarının üzerinde ve ötesinde ölçeklenirken verileri tutarlı ve kullanılabilir durumda tutmak.

Çok daha fazla potansiyel hata vakasına sahip daha birçok "hareketli parça", zarif bozunma yaklaşımlarını kullanmaya zorlar.

Artan dağıtım birimleri sayısı ve altyapıdaki bağımlılıkları, sürdürülmesi gereken karmaşık yapılandırmalara yol açar.

Uygulama ekipleri arasında çabaların tekrarlanması ve birden fazla farklı teknoloji aracılığıyla artan maliyet.

Yalnızca ayrı mikro servis ekipleri olduğunda entegre testler zordur.

Geliştirme ekibinizden daha fazla hareketli parça aracılığıyla daha fazla işlem karmaşıklığı ve daha fazla operasyonel beceri gerekir.

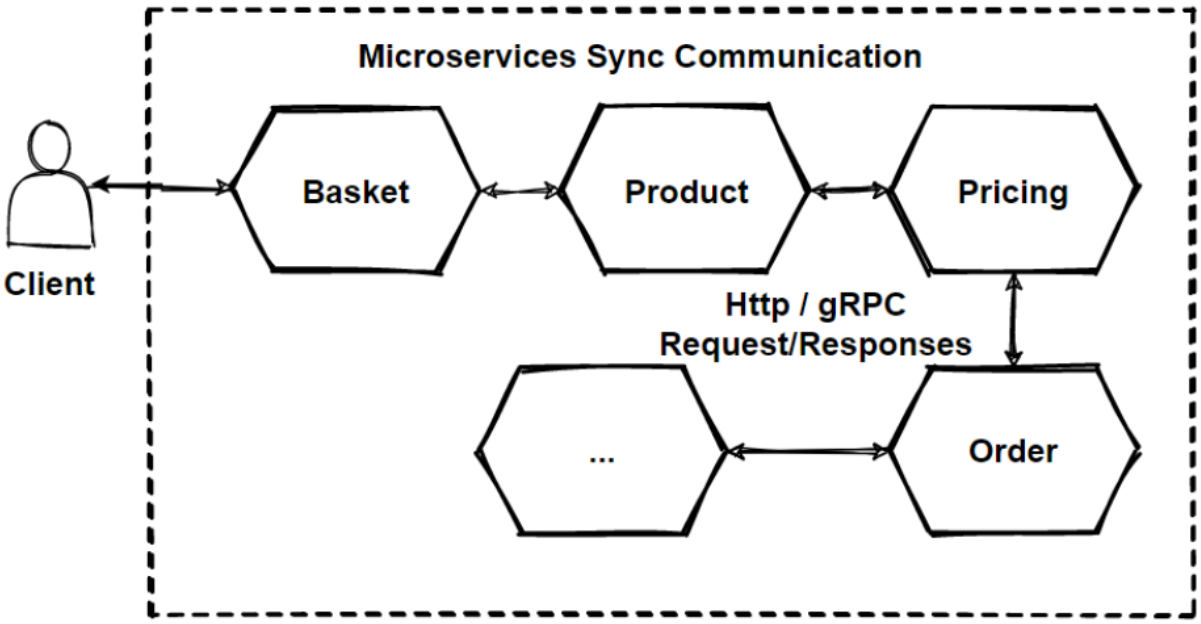
10-How independent microservices communicate with each other?

Mikro servisler, bağımsız olarak geliştirilen ve dağıtılan hizmetlere karmaşık bir yapı olduğundan, iletişim türlerini değerlendirirken dikkatli olmalıyız ve bunları tasarım aşamalarında yönetmeliyiz.

Mikroservis iletişimlerimizi tasarlamadan önce iletişim stillerini anlamalıyız, bunları iki eksende sınıflandırmak mümkündür. İlk adım, iletişim protokolünün senkron veya asenkron olduğunu tanımlamaktır.

What is Synchronous communication ?

Aslında, Senkron iletişimin, senkronizasyon yanıtını döndürmek için HTTP veya gRPC protokolünü kullandığını söyleyebiliriz. İstemci bir istek gönderir ve hizmetten bir yanıt bekler. Bu, yanıt sunucudan gelene kadar istemci kodunun iş parçacığını engellediği anlamına gelir.



Senkronize iletişim protokolleri HTTP veya HTTPS olabilir.

Senkron iletişimde, istemci http protokollerini kullanarak istek gönderir ve servisten yanıt bekler.

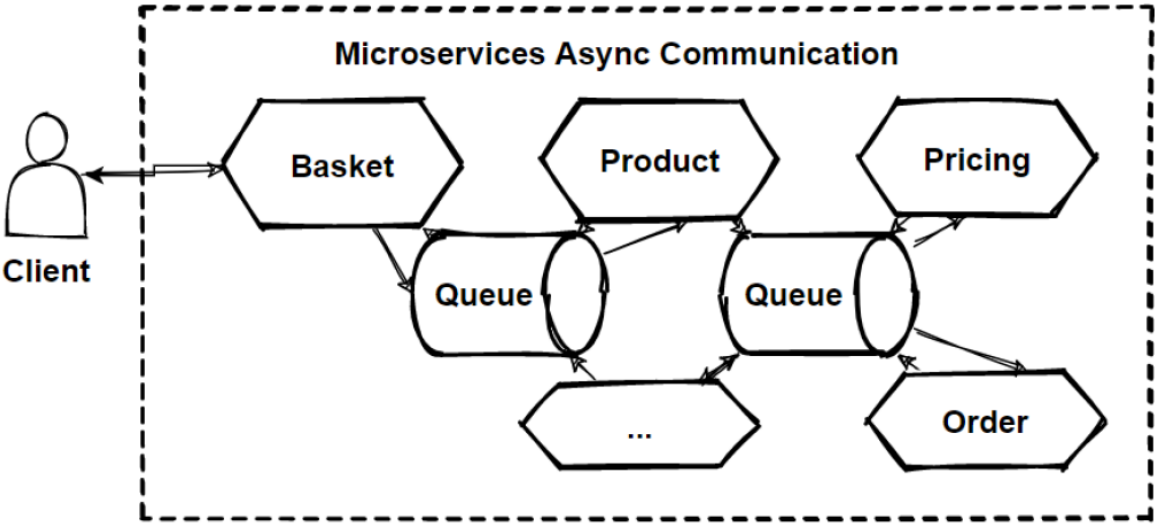
Bu, istemcinin sunucuyu aradığı ve istemcinin işlemlerini engellediği anlamına gelir.

İstemci kodu, HTTP sunucusu yanıtını aldığında görevine devam edecektir. Yani bu işleme Senkron iletişim denir. Bu yolu seçtiğimizde göz önünde bulundurmamız gereken artıları ve eksileri var.

What is Asynchronous communication ?

Temel olarak, Asenkron iletişimde, istemci bir istek gönderir ancak hizmetten bir yanıt beklemez. Yani buradaki kilit nokta, istemcinin yanıt beklerken bir ileti dizisini engellememesi gerektiğidir.

Bu Asenkron iletişim için en popüler protokol AMQP'dir (Gelişmiş Mesaj Sıralama Protokolü). Böylece AMQP protokollerini kullanarak, istemci mesajı Kafka ve RabbitMQ kuyruğu gibi mesaj komisyoncu sistemlerini kullanarak gönderir. Mesaj üreticisi genellikle bir yanıt beklemez. Bu mesaj abone sistemlerinden zaman uyumsuz bir şekilde tüketilir ve kimse aniden cevap beklemez.



Asenkron bir iletişim de uygulamaya göre 2'ye bölünür. Eşzamansız sistemler One-to-one (queue) modunda veya One-to-many (topic) modunda uygulanabilir.

One-to-one(queue) uygulamada tek üretici ve tek alıcı vardır. Ancak One to Many (topic) uygulamasında Çoklu alıcılar vardır. Her istek sıfırdan birden çok alıcıya işlenebilir. One-to-many (topic) iletişimleri zaman uyumsuz olmalıdır.

11-What do you mean by Domain driven design ?

Nesne yönelimli programlama açısından, yazılım kodunun yapısı ve dilinin (sınıf adları, sınıf yöntemleri, sınıf değişkenleri) iş alanıyla eşleşmesi gerektiği anlamına gelir. Örneğin, bir yazılım kredi başvurularını işliyorsa, LoanApplication ve Customer gibi sınıflara ve AcceptOffer ve Withdraw gibi yöntemlere sahip olabilir.

- DDD, uygulamayı gelişen bir modele bağlar.
- Etki alanına dayalı tasarım aşağıdaki hedeflere dayanır:
- projenin birincil odağını çekirdek etki alanına ve etki alanı mantığına yerleştirmek;
- karmaşık tasarımları bir alan modeline dayandırmak;
- Belirli alan sorunlarını ele alan kavramsal bir modeli yinelemeli olarak iyileştirmek için teknik ve alan uzmanları arasında yaratıcı bir işbirliği başlatmak.

Etki alanı güdümlü tasarımın eleştirileri, geliştiricilerin modeli saf ve yardımcı bir yapı olarak sürdürmek için tipik olarak çok fazla izolasyon ve kapsülleme uygulaması gerektiğini savunuyor. Etki alanına dayalı tasarım, sürdürülebilirlik gibi avantajlar sağlarken, Microsoft bunu yalnızca modelin etki alanı hakkında ortak bir anlayış formüle etmede açık avantajlar sağladığı karmaşık etki alanları için önerir.

12-What is container in Microservices ?

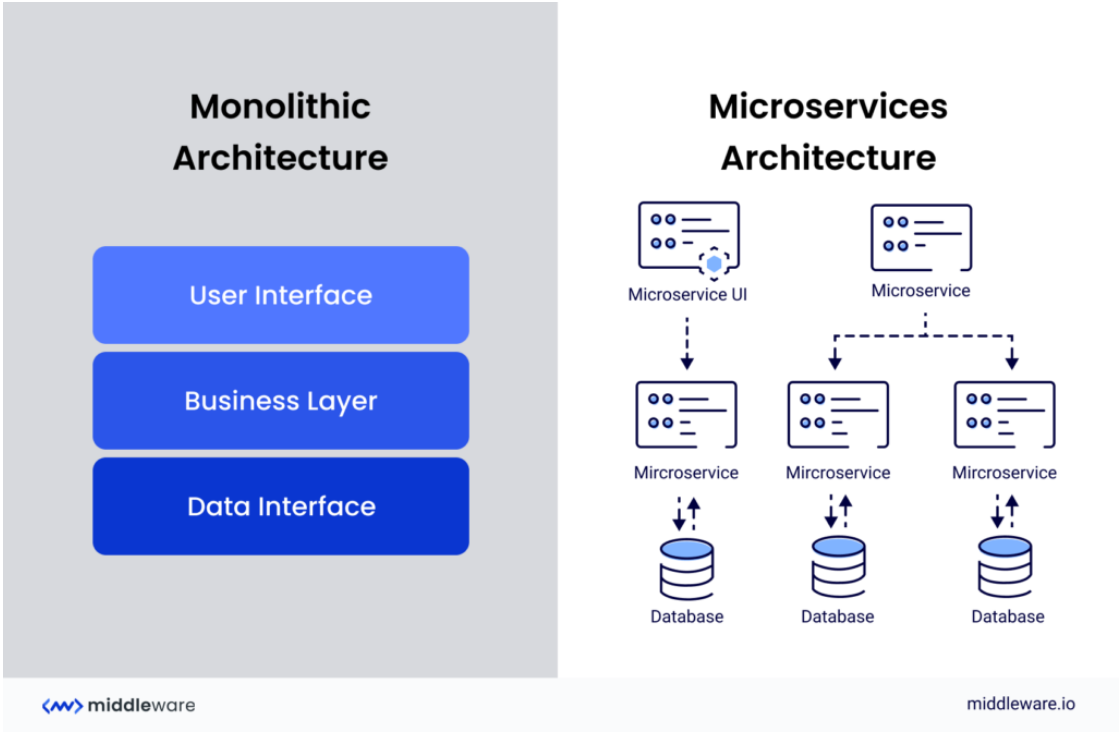
Konteynerler, bir işletim sistemi sanallaştırma biçimidir. Küçük bir mikro hizmet veya yazılım sürecinden daha büyük bir uygulamaya kadar her şeyi çalıştırmak için tek bir kapsayıcı kullanılabilir. Bir kapsayıcının içinde gerekli tüm yürütülebilir dosyalar, ikili kod, kitaplıklar ve yapılandırma dosyaları bulunur. Ancak, sunucu veya makine sanallaştırma yaklaşımlarıyla karşılaştırıldığında, kapsayıcılar işletim sistemi görüntüleri içermez. Bu, önemli ölçüde daha az ek yük ile onları daha hafif ve taşınabilir hale getirir. Daha büyük uygulama dağıtımlarında, birden çok kapsayıcı bir veya daha fazla kapsayıcı kümesi olarak dağıtılabilir. Bu tür kümeler, Kubernetes gibi bir kapsayıcı düzenleyici tarafından yönetilebilir.

13-What are the main components of Microservices architecture ?

1. Clients
2. Identity Providers
3. API Gateway
4. Messaging Formats
5. Databases

6. Static Content
7. Management
8. Service Discovery

14 - How does a Microservice architecture work?



Yukarıda Mikroservis ve Monolithic mimarinin farklarını görebilirsiniz.

İstemci, istek oluşturmak için kullanıcı arabirimini kullanabilir. Aynı zamanda, istenen görevi gerçekleştirmek için API ağ geçidi aracılığıyla bir veya daha fazla mikro hizmet devreye alınır. Sonuç olarak, mikro servislerin bir kombinasyonunu gerektiren daha büyük karmaşık sorunlar bile nispeten kolay bir şekilde çözülebilir.