HW#6

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

Manuel Test	**Otomasyon Testi**
Manuel testi kullanarak, uygulamayı farklı İşletim sistemlerinde test etmek zor olabilir.	Otomasyon testi yardımı ile uygulamayı farklı İşletim sistemlerinde kolayca test edebiliriz.
Test senaryoları manuel olarak yürütülür.	Test senaryoları araçlar yardımıyla yürütülür.
Güvenilirlik daha azdır.	Güvenilirlik daha fazladır.
Daha az maliyetlidir.	Daha pahalıdır.
Bazı test durumları için zaman harcar.	Makine olduğu için davaları yürütmek daha az zaman alır.
İnsan hata yapabilir ve dolayısıyla doğruluk daha azdır.	Makine neredeyse hiç hata yapmıyor (Eğer bunu yapması istenmişse).
İnsan müdahalesini içerdiğinden, uygulamaya erişim kolaylığını kontrol etmekte fayda vardır.	Kullanılabilirliği veya erişilebilirliği kontrol edemeyen araçlar içerir.
Bazen tüm test senaryolarını yürütmek zorlaşır ve test kapsamını etkiler.	Otomasyon testinde, test kapsamı hedefine ulaşabiliriz.
Manuel için, uygulamayı farklı tarayıcılarda test etmek zor olabilir.	Otomasyon, yazılımı farklı tarayıcılarda test etme avantajı sağlar. Selenium grid, uygulamayı farklı tarayıcılarda test etmemize izin verir.
Bu durumda, sisteminizin önüne oturmanız ve insan müdahalesini içerdiği için test senaryoları yürütmeniz gerekir.	Otomasyon komut dosyalarını çalıştırmanız yeterlidir, bir gecede çalıştırabilirsiniz!
Bu testte kendi başınıza raporlar oluşturmanız gerekir.	Burada araç, test senaryosu yürütme raporu oluşturacaktır. TestNG, sizin için rapor oluşturacak çerçevedir.

2 – What does Assert class ?

Assert anahtar kelimesi Java 1.4'ten beri Java programlama dilinin bir özelliğidir. Assertion , geliştiricilerin hataları gidermek ve düzeltmek için programlarındaki varsayımları test etmelerini sağlar. Assert deyimi çok kullanılmayacak nadir hataları Throwable sınıfından fırlatmak yerine kullanılacak en iyi test aşaması olağan durumu yakalama biçimidir.Hata oluşması az olan Exceptionlar programları çok yavaşlatır.Bundan ötürü assert kelimesini kullanıp test aşamasında sadece önlenebilecek hataları bulmak daha mantıklıdır.

ASSERT İFADESİNİN SÖZDİZİMİ

Bir assert ifadesinin sözdizimi şu şekildedir (kısa versiyon):

Assert *expression*;

Assert *expression1* : *expression2* ;

assert age>18 :"Artık bir yetişkinsiniz";

- *Expression1* bir boolean ifadesi olmalıdır.
- *expression2* bir değer (void – dönüş olmamalıdır) döndürmesi gerekir.

Assert Açıklamada, şunlar çalışıyor:

-
    - Iddia etkinleştirilirse, assert ifadesi değerlendirilecektir. Aksi halde yok edilmez.
    - Eğer *expression1* değerlendirilir ve yanlışsa , AssertionError hata programı hemen durur nedeni atılır. Ve ifadenin varlığına bağlı *olarak* :
-
        - Eğer *expression2* yoksa, o zaman AssertionError ayrıntı hata mesajı ile atılmaz.
        - Eğer *expression2* mevcutsa, daha sonra bir dize gösterimi *expression2* 'ın dönüş değeri detaylı hata mesajı olarak kullanılır.
    - Eğer *expression1* değerlendirmesi doğru ise program normal olarak devam eder.

3 - How can be tested 'private' methods ?

Bir uygulama geliştirken bir kod biriminin test edilebilirliğinin önündeki engel OOP prensipleri olmamalıdır. Eğer öyle ise, OOP prensipleri yanlış uygulanmış demektir. OOP kod geliştirmek için kullanılan bir araçtır. *private* yerine göre kullanılabilecek bir erişim mekanizmasıdır. Lakin *private* olan bir metot benim için yoktur, yani test edilmesi mümkün değildir. Benim için test edilemeyen kod birimi olamayacağı için, *private* ile tanımlanmış metotları yok sayarım. Test edilemeyen bir metodun başına her türlü iş gelebilir. Benim bu tür metotları görünce ilk yaptığım şey, metodu

hemen *protected* yapmak ve o metodu izole bir şekilde test eden bir birim testi yazmak olur. *Protected* yetmedi ise *public* yaparım. Nihai amaç metodu bir şekilde test edebilmektir.

Ne test edilir, ne test edilmez, nasıl test edilir tartışmalarının sonu gelmez ne yazık ki. Geçenlerde yine kocaman, ne yedüğü belli olmayan bir sınıfa yeni bir özellik eklemem istendi. Eğer gerekli kodu o sınıfa direk ekleseydim, yeni eklediğim kod birimini test etmem çok zor olacaktı, bunu biliyordum. Sınıfın tümünü test edecek bir birim testi oluşturmak için vaktim yoktu. Bende bunun yerine yeni bir static iç sınıf oluşturup, kodu bu sınıfın bir metodu olarak geliştirdim ve bir birim testi ile test ettim. Testin başarılı olduğunu gördükten sonra ne yedüğü belirsiz diye tabir ettiğim sınıfın gerekli metoduna oluşturduğum yeni sınıfın bir nesnesini yerleştirerek, static sınıfın metodunu orada koşturdum. Böylece ne yedüğü belirsiz sınıf dolaylı bir şekilde yeni oluşturduğum kod birimine sahip oldu. Doğal olarak gelen ilk soru şu oldu: "*neden static bir iç sınıf oluşturdun ki? Orada bu işi görecek bir metot vardı, o metoda kendi kodunu ekleyebilirdin*". Eklerdim, eklemesine, lakin bu sınıfa eklediğim kodu nasıl test edebilirdim? Edemezdim. Sınıf için daha önce hiç birim testi yazılmamıştı. Sınıfa eklediğim kod birimini test edebilmek için tüm uygulamayı ayağa kaldırıp, benim eklediğim kod birimi çalışıyor mu diye otomatize edilemeyecek türden bir test yapmam gerekirdi. Yok efendim. bu tür yöntemler sınıf enflasyonuna sebep oluyormuş. Olursa, olsun, sorun nedir? Önemli olan benim yeni kod birimini izole bir şekilde test etmemdi.

OOP, tasarım şablonları ya da tasarım prensipleri sadece test edilebilirliğin emrinde çalışabilecek erlerdir, tersi değil! Bu saydığım araç, gereçlerin hepsinin efendisi test edilebilirlik özelliğidir. Bunların arkasına saklanıp, bu kod birimi test edilemez diyen yazılımcı kendisine ve müşterisine ihanet içinde olur.

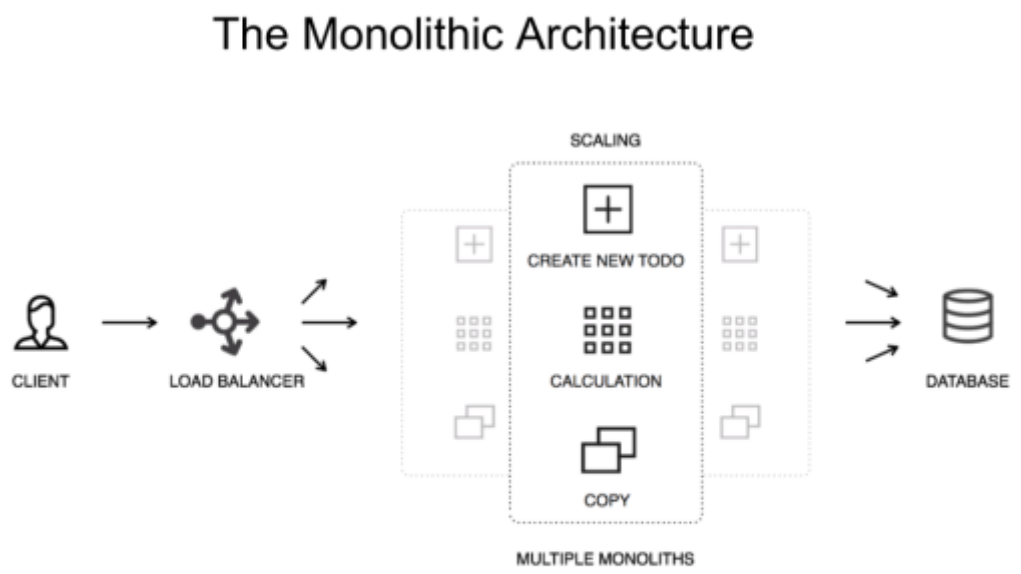
4 – What is Monolithic Architecture ?

Monolithic architecture yazılımın **self-contained**(kendi kendine yeten) olarak tasarlanması anlamına gelmektedir. Bir standart doğrultusunda "tek bir parça" olarak oluşması da diyebiliriz. Bu mimarideki component'ler **loosely coupled** olmasından ziyade, **interdependent** olarak tasarlanmaktadır.

Günümüze baktığımızda kurumsal projeler, Service Oriented Architecture(SOA) ile geliştirilmeye başlanmış ve büyük ölçüde yerini zaten almış durumdadır. Geleneksel SOA mimarisinde geliştirilen tüm component'lerin de, tek bir çatı altında olduğunu da görmekteyiz. Çünkü yakın geçmişten bu yana SOA ile birlikte **manageability**(yönetilebilirlik), **maintenance**(bakım) ve **interoperability**(birlikte çalışabilirlik) gibi kavramlar göz önüne alınmıştır. Günümüzdeki şirketlerin IT yaklaşımlarına baktığımızda ise genelde **IT for Business** kapsamında olduğundan dolayı, her zaman pazarlama odaklı gitmektedirler. Bu doğrultuda sürekli artan bir entegrasyon ihtiyaçları doğmaktadır. Bu bitmeyen ihtiyaçlar doğrultusunda ise Monolithic architecture ile tasarlanmış olan SOA'lar, gitgide istemsizce büyümektedirler.

Bu noktaya kadar her şey "büyüme" haricinde normal görünebilir fakat problem nerede/ne zaman başlıyor? İşte bu soruya geçmeden önce Monolithic architecture'ın dezavantajlarını bir ele alalım.

Monolithic Architecture'ın Getirdiği Bazı Dezavantajlar



Yukarıdaki resme baktığımızda bölünmez, self-contained olarak tasarlanmış monolithic bir yapı görüyoruz. Scaling için bir load balancer arkasına koyulmuş fakat bu durumda da scale edilmek istenin component'in aksine, tüm monolith yapının kopyasını farklı ortamlarda saklamak durumunda kalınıyor.

Diğer dezavantajlarını maddelemek gerekirse:

- Tüm component'lerin aynı framework, aynı programlama dili ile geliştirilmesinin gerekmesi
- Bir component üzerinde olan değişiklik için, tüm monolith yapının tekrar deploy edilmesi ve restart edilmesi durumunda kalınması
- Versiyon yönetiminin gitgide zorlaşması
- Birbirlerine olan bağımlılıklarından dolayı, bir component için yapılan değişimden diğer component'in etkilenebilmesi
- Continuous Delivery'nin uygulanmasının zorlaşması

Bu dezavantajların bazıları monolithic mimarinin büyümesi ile gelmese de en major problemlerden birtanesi, monolithic mimari üzerindeki component'lerden herhangi birinde olan değişikliğin deployment'ı yapıldığında, bu durumdan diğerlerinin de etkilenebiliyor/etkilenebilecek olmasıdır.

Düşünelim. Belediye otobüslerine hangi durakta olduklarını ve her durak içinde ilgili otobüsün gelmesine tahmini kalan sürenin gösterimini yapan ekranların servisini geliştiriyoruz. Bizden otobüs içerisindeki ekranda gösterilmesi gereken bazı yeni özellikler istendi. İlgili geliştirmeyi ilgili ekip yaptı ve deployment'ı gerçekleştirdi. Peki ya diğer servis olan duraklardaki otobüs sürelerini gösteren fonksiyonun geliştirilmesinde yarım kalan bir şey varsa? Veya otobüs içerisindeki ekranlar için geliştirilen serviste bir hata oluşursa ve diğer servise olan bağımlılığından dolayı, her iki serviste kullanılamaz hale gelirse? Bu ve bunun gibi farklı varyasyon ve senaryolar göz önüne alındığında,

monolithic mimaride geliştirilen servislerde yazılım ekiplerinin birbirleri ile iletişim becerilerinin yüksek olması gerektiği, farklı özellikler geldikçe code base'in daha da karmaşıklaşacağı ve micro deployment'lar yapılamayacağı görülüyor. Yani buradaki tek problemimiz scale etmek ve micro deployment'ları sağlayabilmek değil.

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

- Her bir fonksiyon için test yazmak doğru değildir. Sadece sistemin davranışını etkileyen birimler için test yazmak daha uygundur. Örneğin getter and setter metot'ları için test yazmamak.
- Birim Test otomatize edilebilecek şekilde olmalıdır(CI/CD).
- Her bir logic için ayrı bir test metot'u yazılmalıdır(one test per logical path).
- Sadece mantık(bussiness logic) içeren birimler için değil aynı zamanda birimin performansını ölçmek içinde Birim Test yazılmalıdır.
- Test'i yapılan birimde güncelleme yapıldığında(bussiness logic hariç), birimin test'inin güncellemesine gerek olamayacak şekilde yazılmış olması gerekir.
- Private method'lar için birim test yazmak gerekmez. Public method'lar için Birim Test yazılması yeterlidir.(Public method'lar scope'ları içerisinde private method'ları çağıracağı için private method'lar dolaylı yoldan test edilmiş olurlar.)
- Birim Test yazılırken **Arrange-Act-Assert** veya **Given-When-Then** gelenekleri(convention) yaygın kullanılmaktadır.
- Önce test yazılmalıdır. Sonra testi geçebilecek(success) birim yazılmalıdır.
- Kaynak kod ile test kod'unu izole etmek gerekir.(Kaynak kod'lar -> **src/main**, Birim Test kod'ları **src/test**)
- Hata(bug) çıktığında düzeltmeden(fix) önce testini sağlamlaştırıp(expose test) hata düzeltmek yapmak birim testing güvenilirliğini arttıracaktır.
- Testleri bir birinden **bağımsız** yazmak tercih edilmelidir. Örneğin database ile ilgili bir sınıfın testleri yazılırken doğrodan database bağlantısı oluşturmak yerine abstract sınıflar üzerinden veya mock database connection nesnesi(**mock objects fill missing parts**) üzerinden bağlantı sağlanmalıdır.
- Yazılan test, tüm olası durumları içermelidir(**cover all the paths**).
- Test metot'larının ismi okunabilir(readable) ve testi yapılan birimi kolay takip edilebilecek şekilde olmalıdır(self descriptive).
- Test script'lerinde yapılan değişiklikleri gözlemlemek için versiyon kontrol sistemi kullanılmalıdır.
- Hata yakalama(exception handling) durumlarını test etmek için test metot'u içerisinde Try-Catch blok'u kullanılmamalıdır(annotation kullanılabilir).
- POJO test edilmemelidir(don't test java).

- Test metot'u gittikçe karmaşıklaşıyorsa(ortalama 5–10 satır ideal), mock object sayısı çok fazlaysa(en fazla 4 mock object ideal) kodun yeniden yapılandırması gerekmektedir.
- Her bir bug bulunduğunda yeni bir **Birim Test** eklenmelidir.
- Test yazma sürecini hızlandırmak için tool kullanmak. Örneğin java için JUnit framework'ünü kullanarak **Birim Test** yazmak.
- Konfigürasyon ve log alma işlemleri test edilmemelidir.
- Test kurulum aşaması(setup) mümkün olduğunca az kod satırından oluşmalıdır. Test sınıfının kod satır sayısı kaynak kod'taki sınıfı geçmemelidir.
- Aynı test metodu içerisinde birbirinden bağımsız durumu(case) test eden birden fazla Assert kullanılmamalıdır.
- Aynı durumu(case) test eden yapı birden fazla test metot'u ile bağlantılı olmamalıdır. Aksi halde kaynak kod değiştiğinde tüm test metot'larınında değişmesi gerekir.

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, Actuator, 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.

Actuator, 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 ?

Geliştirilmiş Ölçeklenebilirlik-: Her mikro hizmetin ücretsiz çalıştırma rolü, tek bir mikro hizmeti eklemeyi, ayıklamayı, güncelleştirmeyi ve ölçeklendirmeyi nispeten kolaylaştırır. Bu, uygulamayı içeren diğer mikro hizmetleri rahatsız etmeden tamamlanabilir. Aracı, daha fazla işlem gücü sağlayarak kişisel mikro hizmetleri otomatik olarak ölçeklendirebilir. Ayrıca, gerektiğinde mikro hizmetlerin yeni örneklerini döndürmeye de yardımcı olabilir. Netflix, ölçeklendirme sorunlarının üstesinden gelmek için Mikro Hizmetleri kullanmanın mükemmel bir örneğidir.

Daha İyi Hata Çözümü-: Mikro hizmet mimarisinin avantajları, içerik oluşturucuların basamaklı hataları özelliklerle denetlemesine olanak tanır. Bir arama hizmetinin hiçbir zaman yanıt vermeyen başarısız bir hizmet için beklemede kalması gerekiyorsa, sunucu kaynak sıkıntısını önlemeye yardımcı

olur. Mikro Hizmetler sırasında Uzaktan Yordam Çağrılarından (RPC) kaçınmalıyız çünkü basamaklı hataya neden olurlar. Bir işlemi ayrı bir hizmete yeniden yapılandırırsanız, içeren işlemi tamamen zaman uyumsuz olacak şekilde yeniden tasarlarlar.

Programlama Dili ve Teknoloji Platformu-: Oluşturucu, mikro hizmet programlarını herhangi bir dilde bağlayabilir. Ayrıca, mikro hizmetlerin herhangi bir platformda çalışmak üzere bağlanmasına yardımcı olurlar. Bu, görevin ihtiyacına ve ekibinizin becerilerine uyan programlama dillerini ve teknolojilerini çalıştırmak için daha fazla esneklik sağlar. Her grup, hizmetlerini yürütmek için diğer teknolojileri tercih edebilir. Bulut tabanlı mikro hizmetlerin kullanıcısı, uygulamaya internete bağlı herhangi bir cihazdan erişebilir.

Daha İyi Veri Güvenliği ve Uyumluluklar: Mikro hizmetlerin her avantajı, içindeki hassas verileri yönetir ve savunur. Oluşturucu mikro hizmetler arasında veri bağlantısına başladığında, bilgilerin güvenliği en önemli endişe kaynağıdır. Mikro hizmetleriniz gizli bilgileri yönetiyorsa, güvenli bir API geliştiricilere tam denetim sağlar. Ayrıca, büyük uygulama ve onu kullananlar için ne tür verilere erişilebildiğini bilmeye yardımcı olur.

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

Mikro hizmet mimarisi eski sistemden daha karmaşıktır. Mikro hizmet ortamı daha karmaşık hale gelir, çünkü ekip birçok hareketli parçayı yönetmek ve desteklemek zorundadır. Bir kuruluşun mikro hizmet yolculuğunda karşılaştığı en önemli zorluklardan bazıları şunlardır:

- Sınırlı Bağlam
- Dinamik Ölçeği Artırma ve Küçültme
- Izleme
- Arızaya dayanıklılık
- Döngüsel bağımlılıklar
- DevOps Kültürü

Sınırlı bağlam: Sınırlı **bağlam** kavramı, Etki Alanı Odaklı Tasarım (DDD) çevrelerinden kaynaklanır. Hizmete ilk Nesne modeli yaklaşımını teşvik eder, hizmetin sorumlu olduğu ve bağlı olduğu bir veri modeli tanımlar. Sınırlı bir bağlam, modele yönelik belirli sorumluluğu açıklığa kavuşturur, kapsüller ve tanımlar. Etki alanının dışarıdan dikkatinin dağılmamasını sağlar. Her modelin bir alt etki alanı içinde örtülü olarak tanımlanmış bir bağlamı olmalıdır ve her bağlam sınırları tanımlar.

Başka bir deyişle, hizmet verilerine sahiptir ve bütünlüğünden ve değişkenliğinden sorumludur. Mikro hizmetlerin en önemli özelliği olan bağımsızlık ve ayırmayı destekler.

Dinamik ölçeği artırma ve azaltma: Farklı mikro hizmetlerdeki yükler, türün farklı bir örneğinde olabilir. Mikro hizmetinizin ölçeğini otomatik olarak artırmanın yanı sıra ölçeği de otomatik olarak küçültmelidir. Mikro hizmetlerin maliyetini azaltır. Yükü dinamik olarak dağıtabiliyoruz.

İzleme: Geleneksel izleme yöntemi, mikro hizmetlerle iyi uyum sağlamaz, çünkü daha önce tek bir uygulama tarafından desteklenen aynı işlevselliği oluşturan birden fazla hizmetimiz vardır. Uygulamada bir hata ortaya çıktığında, kök nedeni bulmak zor olabilir.

Hata **Toleransı: Hata** toleransı, genel sistemi çökertmeyen bireysel hizmettir. Uygulama, hata oluştuğunda belirli bir memnuniyet derecesinde çalışabilir. Hata toleransı olmadan, sistemdeki tek bir arıza tam bir arızaya neden olabilir. Devre kesici hata toleransı elde edebilir. Devre kesici, isteği harici servise saran ve arızalı olduklarını algılayan bir desendir. Mikro hizmetlerin hem iç hem de dış hataları tolere etmesi gerekir.

Döngüsel Bağımlılık: Farklı hizmetler arasında bağımlılık yönetimi ve işlevselliği çok önemlidir. Döngüsel bağımlılık, tanımlanmaz ve derhal çözülmezse bir sorun oluşturabilir.

DevOps Kültürü: Mikro hizmetler, **DevOps'a** mükemmel uyum sağlar. Daha hızlı teslimat hizmeti, veriler arasında görünürlük ve uygun maliyetli veriler sağlar. Konteynerleştirme anahtarı kullanımlarını Hizmet Odaklı Mimari'den (SOA) Mikro Hizmet Mimarisi'ne (MSA) genişletebilir.

Mikro hizmetlerin diğer zorlukları

- Daha fazla mikro hizmet ekledikçe, birlikte ölçeklendirilebildiklerinden emin olmalıyız. Daha fazla ayrıntılılık, karmaşıklığı artıran daha fazla hareketli parça anlamına gelir.
- Geleneksel günlük kaydı etkisizdir, çünkü mikro hizmetler durum bilgisi yoktur, dağıtılmış ve bağımsızdır. Günlüğe kaydetme, çeşitli platformlardaki olayları ilişkilendirebilmelidir.
- Daha fazla hizmet birbiriyle etkileşime girdiğinde, başarısızlık olasılığı da artar.

10 - How independent microservices communicate with each other?

Mikro hizmetler, herhangi bir kuruluş için harika bir çerçevedir, ancak birbirleriyle sorunsuz bir şekilde iletişim kuramazlarsa hiçbir faydası yoktur. Mikro hizmetlerin ayrıntılarını ve bunların bulut yerel uygulamalarıyla nasıl ilişkilendirildiğini inceleyelim. Her kuruluş, ölçeklenebilir uygulamalar oluşturmaya önem verir. Bulutta yerel yaklaşımın ölçeklenebilirlik zorluklarının üstesinden gelmek için en iyisi olduğu bir gerçektir. Bulut yerelinin temel ilkelerinden biri mikro hizmetlerdir.

11 - What do you mean by Domain driven design ?

Etki alanına dayalı tasarım (DDD), onu kullananların etki alanı veya bilgi alanı etrafında odaklanan bir yazılım geliştirme felsefesidir. Yaklaşım, ihtiyaç duyanların karmaşık gereksinimlerine odaklanan ve gereksiz hiçbir şey için çaba harcamayan yazılımların geliştirilmesine olanak tanır.

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.

13 - What are the main components of Microservices architecture ?

1. Microservices

2. Containers

3. Service mesh

4. Service discovery

5. API gateway

14 - How does a Microservice architecture work?

Günümüzde mikro hizmet mimarilerinin avantajlarını anlamak için her şeyin nereden başladığını anlamak çok önemlidir.

Monolitik uygulamalar

Başlangıçta tek bir sunucuda bulunan her uygulama üç katmandan oluşuyordu:

Sunum

Uygulama/iş mantığı

Veri tabanı

Bu katmanlar, bir veri merkezindeki tek, yekpare bir sunucuda bulunan tek, iç içe geçmiş bir yığın halinde oluşturulmuştur. Bu model, her sektör dikeyinde ve teknoloji mimarisinde yaygındı. Genel olarak konuşursak, bir uygulama, belirli bir işleve hizmet eden bir kod modülleri topluluğudur - örneğin, bir veritabanı, çeşitli iş mantığı türleri, grafik işleme kodu veya günlük kaydı.

Bu monolitik mimaride kullanıcılar, iş mantığı katmanı ve veritabanı katmanıyla konuşan sunum katmanıyla etkileşime girdi ve daha sonra bilgi yığından son kullanıcıya geri gitti. Bu, bir uygulamayı düzenlemenin etkili bir yolu olsa da, bir donanım hatası veya kod hatası olması durumunda uzun kesintilere neden olabilecek birçok tek hata noktası oluşturdu. Ne yazık ki bu yapıda “kendini iyileştirme” yoktu. Sistemin bir parçası hasar görmüşse, bir donanım veya yazılım düzeltmesi şeklinde insan müdahalesi ile onarılması gerekir.

Ayrıca, bu katmanlardan herhangi birinde ölçeklendirme yapmak, tamamen yeni bir sunucu satın almak anlamına geliyordu. Tek bir sunucu üzerinde çalışan monolitik bir uygulama satın almanız ve kullanıcıların bir kısmını yeni sisteme ayırmanız gerekiyordu. Bu segmentasyon, gecelik toplu raporlarla uzlaştırılması gereken kullanıcı verileri siloları ile sonuçlandı. Neyse ki, web sayfaları ve mobil uygulamalar daha popüler hale geldikçe müşteri ihtiyacı azaldı ve yeni uygulama geliştirme yöntemleri şekillenmeye başladı.