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

Manuel testte, test senaryoları bir insan test cihazı ve yazılım tarafından yürütülürken otomasyon testi, test senaryolarını yürütmek için otomasyon araçlarını kullanır.

Manuel test, zaman alıcıdır ve insan kaynağı gerektirirken otomatik test, manuel bir yaklaşımdan önemli ölçüde daha hızlıdır.

Manuel Testte keşif testi mümkünken Otomasyon rastgele testlere izin vermiyor

Manuel teste yapılan ilk yatırım nispeten daha düşüktür. ROI, uzun vadede Otomasyon testine kıyasla daha düşüktür.

İnsan hataları olasılığı nedeniyle manuel testler o kadar güvenilir değilken otomatik test, araçlar ve komut dosyaları tarafından gerçekleştirildiği için güvenilir bir yöntemdir.

UI'da yapılan değişiklik manuel bir test cihazının yürütülmesini engellemez iken AUT'nin kullanıcı arayüzündeki önemsiz bir değişiklik için bile, otomatik test komut dosyalarının beklendiği gibi çalışması için değiştirilmesi gerekir.

Manuel testte performans testi yapılamaz, Yük Testi, Stres Test i vb. gibi performans testlerinin zorunlu olarak bir otomasyon aracı tarafından test edilmesi gerekir.

Manuel testler paralel olarak yürütülebilir, ancak pahalı olan insan kaynağınızı artırmanız gerekirken bu test, otomasyon testi ile farklı işletim platformlarında paralel olarak yürütülebilir ve test yürütme süresini kısaltır.

2- What does Assert class ?

Assert, bir test senaryosunun başarılı veya başarısız durumunu belirlemede kullanılan ve lang.Object sınıfını extends eden bir Junit sınıfıdır.

İçerisinde birçok metot barındırır. Bunların bazılarını tek kelime ile açıklamak gerekirse,

assertEquals: İki nesnenin eşit olup olmadığını kontrol eder.

assertTrue: Bir koşulun doğru olup olmadığını kontrol eder.

assertFalse: Bir koşulun yanlış olup olmadığını kontrol eder.

assertNotNull: Bir nesnenin boş olmadığını kontrol eder.

assertNull: Bir nesnenin boş olup olmadığını kontrol eder.

assertSame: AssertSame() yöntemi, iki nesne başvurusunun aynı nesneyi gösterip

göstermediğini test eder.

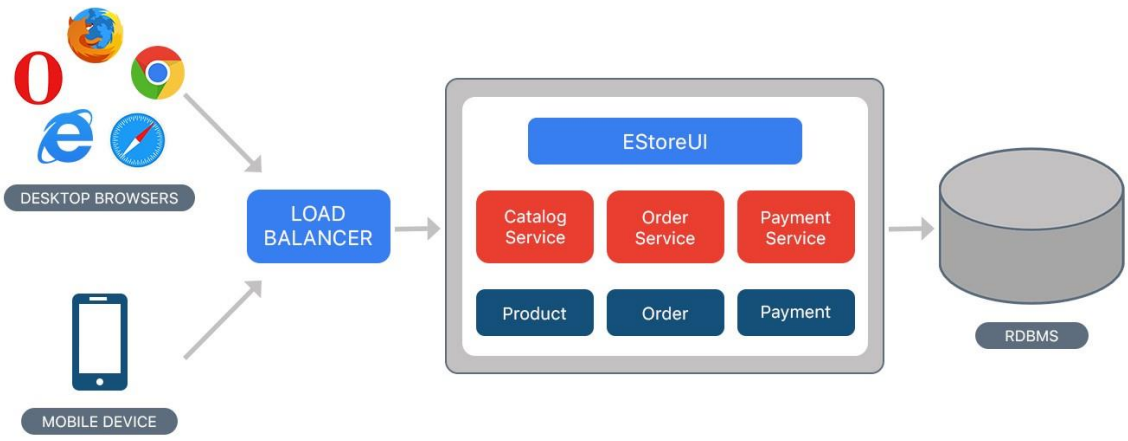
3 - How can be tested 'private' methods ?

Genellikle private metotları doğrudan birim testi yapmayız. Private oldukları için bunları bir uygulama detayı olarak kabul ederiz. Hiç kimse onlardan birini aramayacak ve belirli bir şekilde çalışmasını beklemeyecek.

Bunun yerine genel arayüzünü test ederiz. Private metotlarımızı çağıran yöntemler beklediğimiz gibi çalışıyorsa, uzantı olarak private metotlarımızın da doğru çalıştığını varsayarız.

4 – What is Monolithic Architecture ?

Bu mimari yaklaşım, uygulamanın tüm parçalarının tek bir çatı altında geliştirilmesi ve sunulması olarak tanımlayabiliriz.



Bu monolotik mimariyi inceleyecek olursak, Order Service'inin çok yoğun kullanıldığını ve kaynakların yetmediğini düşünelim. Bu gibi durumlarda ölçeklendirme yapmamız gerekir. Projenin kopyası alınarak bir sunucuya aklatırılır ve load balancer yardımı ile yük dağıtımı gerçekleştirilir. Fakat tüm bileşenler birbirine bağımlı olduğundan bileşenleri ayırarak bir yük dağılımı değil de tüm projenin kopyası alınarak bir ölçeklendirme gerçekleştirilmek zorundadır.

Bu şekilde bir avantaj, dezavantaj ayırımı yapacak olursak,

Geliştirmesi basit bir mimari yapıdır. Test edilebilirliği kolaydır,

Deployment oldukça kolaydır. Ölçeklendirme oldukça kolaydır. Yatay ölçeklendirme ile kopyasını Load Balancer arkasında çalıştırabiliriz.

Fakat dezavantajları oldukça fazladır.

Bakımı proje büyüdükçe zorlaşır. Uygulama büyüdükçe projeyi bağlama süresi yavaşlar.

Uygulama güncelleneceği zaman tüm uygulamayı tekrardan ayağı kaldırmak gereklidir.

Odaklı ölçeklendirmeye imkan vermiyor.

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

Unit testler olabildiğince küçük kod parçaları olmalılar. Hata veren bir unit test çok hızlı bir şekilde okunabilmeli, testteki veri mi yoksa test edilen kod mu sıkıntılı çabucak anlaşılabilir olmalıdır.

Unit testi "unit" yapan şey sadece ve sadece test ettiği kodu test ediyor oluşudur. Test edilen kodun herhangi bir bağımlılığı varsa test sırasında bu bağımlılık izole edilir. Yani bağımlı olan sınıfın belirli bir davranışı sergilediği varsayılır, simüle edilir. Buna mocking denir ve birçok dilde bunu yapmamızı sağlayan kütüphaneler vardır.

Unit testler asla birbirine bağımlı olmamalıdır. Bir unit testin çalışması için öncesinde başka bir testin başarıyla çalışmış olması gerekmemelidir. Böyle bir bağımlılık testlerin sıralı çalıştırılmasını zorunlu kılar ve çoğu test suite yazılımı (junit, vb.) test senaryolarının sıralı çalışacağını garanti etmez.

Son derece normal bir şekilde test ettiğimiz fonksiyonun birden fazla dalı olabilir. Farklı girdilerde farklı çıktılar üreten değişik iş akışlarına sahip olabilir. Hata veren testlerden koddaki hatayı daha hızlı anlayabilmemiz için bir testin sadece tek bir şeyi test ediyor olması gerekmekte.

Unit testleri hızlıca çalıştırılıp sonuç alabileceğimiz şekilde yazmalıyız. Uzun test çalışma süreleri genellikle geliştiricilerin testleri çalıştırmadan build almasıyla sonuçlanıyor.

Test edeceğimiz kodun ne yapması gerekiyor? Asıl varoluş amacı ne? Bu soruya vereceğimiz en basit ve hızlı cevap için hemen bir test yazalım.

Test edilen kodun uç senaryolarını da düşünelim ve mutlaka bu senaryolar için test yazalım.

Test edilen kodda yol ayrımları (if/else, loop, vb.) varsa mutlaka o yol ayrımlarına giren ve girmeyen testleri *ayrı ayrı* yazmaya özen gösterelim.

Eğer kodda belirli bir durumda bir hata oluştuğu iddia ediliyorsa, kodu okuyup bug-fix geliştirmeden önce bu hatayı oluşturan bir unit test yazalım.

Test edilen kodun istisna durumları varsa, Java'da buna Exception deniyor, bunlar için de test yazmalıyız.

Bir test hata vermeye başladığında test kodunu okumadan önce hatanın hangi senaryoda ve hangi durumda olduğunu anlayabiliyor olmalıyız.

Unit testler üzerinde çalıştığı ortamdan bağımsız şekilde her yerde çalışabilecek kadar izole geliştirilmeli.

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

Çünkü prensip olarak hata veren testlerden koddaki hatayı daha hızlı anlayabilmemiz için bir testin sadece tek bir şeyi test ediyor olması gerekmektedir. Buna göre tasarlanmıştır.

7 - What are the benefits and drawbacks of Microservices ?

Avantajları olarak,

- Çok dilli mimariyi desteklemesi,
- Kolay ölçeklendirilebilir olması,
- Daha iyi bir takım yönetimine olanak sağlaması,
- Yenilik yapmak daha kolay. Servis bazında değişiliklik yaparak diğer servisleri etkilemeden teknolojide değişiklikler yapabiliyor olmamız,
- Kendine ait ver, tababına sahip olması,
- Servis odaklı ölçeklendirme yapılabiliyor olması söylenebilir.

Dezavantajları olarak,

- Implemestasyon büyük projeler için envanter tutulmadığı takdirde kolay değildir.
- Debug kolay değildir. Hatanın nerden olduğunu bulmak için Network üzerinde izleme yapılmalı, bunlar için monitoring sistemler geliştirilmeldir.
- Hata yönetimi kolay değildir diyebiliriz.

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

Çalışan Springboot uygulaması hakkında endpointler yardımıyla bilgi (projenin bilgileri, çalışıp çalışmadığı, spring tarafından yönetilen beanler, trafik durumu vb) almamızı sağlar.

Varsayılan endpointlerin birkaçından bahsedesek olursak,

/actuator : Endpoint listesini gösterir.

/actuator/beans : Springboot tarafından yönetilen nesneleri listeler.

/actuator/mappings : Tanımlı @RequestMapping'leri gösterir.

/actuator/httptrace : Http bilgileirni görüntüler. Varsayılan olarak son 100 http isteğini listeler. HttpTraceRepository bean gerektirir.

/actuator/health : uygulamanın durumu hakkında genel bilgi verir (disk veya database erişim durumu vs.)

/actuator /env : Spring' in ConfigurableEnvironment değerlerini listeler

/actuator/metrics : Springboot uygulamamız için metrik verilerini gösterir.

/actuator/dump : thread dump almayı sağlar

Varsayılan endpoint path değerleri yerine istenilen değerler verilebilir.

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

Geleneksel izleme biçimleri, mikroservis tabanlı bir uygulama için çalışmayabilir. Kullanıcı arabiriminden gelen bir isteğin, isteğini yerine getirebilecek olana ulaşmadan önce birden çok servisi geçtiği bir senaryo düşünün. Bu geçişin sonucu, dolambaçlı bir servis yoludur ve uygun izleme araçları olmadan, bir sorunun altında yatan nedeni belirlemek zor olabilir.

Herhangi bir yazılım geliştirme yaşam döngüsünün (SDLC) test aşaması, mikroservis tabanlı uygulamalar için giderek daha karmaşık hale geliyor. Her bir mikroservisin bağımsız doğası göz önüne alındığında, bireysel servisleri bağımsız olarak test etmeniz gerekir.

Ölçeklenebilirlik, mikroservis mimarisiyle ilişkili başka bir operasyonel zorluktur. Mikroservislerin ölçeklenebilirliği genellikle bir avantaj olarak lanse edilse de, mikroservis tabanlı uygulamalarınızı başarıyla ölçeklendirmek zordur.

Optimizasyon ve ölçeklendirme, daha karmaşık koordinasyon gerektirir. Tipik bir mikroservis çerçevesinde, bir uygulama, ayrı sunucularda barındırılan ve dağıtılan daha küçük bağımsız servislere bölünür. Bu mimari, ayrı ayrı bileşenlerin koordine edilmesini gerektirir; bu, özellikle uygulama kullanımında ani bir artışla karşılaştığınızda başka bir zorluktur.

Her servis için gereken hata toleransı. İşletmelerin, mikroservislerinin dahili ve harici arızalara dayanacak kadar esnek olması gerekir. Mikroservis tabanlı bir uygulamada, bir bileşenin arızalanması tüm sistemi etkileyebilir

10 - How independent microservices communicate with each other?

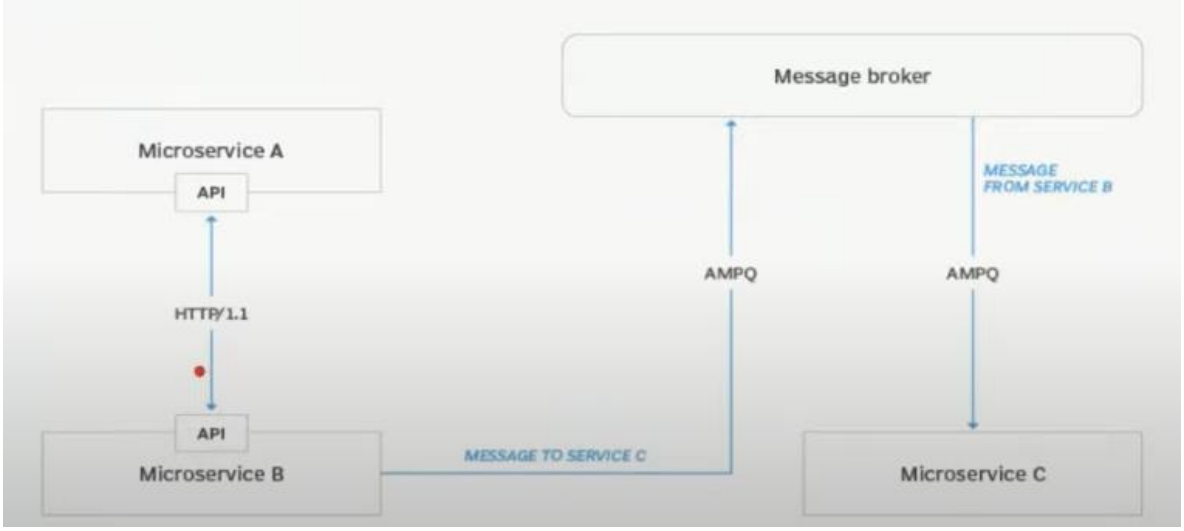
Senkron iletişim

Senkron iletişim kullanılan mikroservisler arasında http ile haberleşme ve call yaptığımızda aslında mikroservis A, mikroservis B den gelecek cevabı bekliyor.
Bu yüzden de mikroservis A'nın response time' ı B ye bağımlı oluyor.

Asenkron iletişim

Mikroservisler birbirlerinden habersiz belli bir message broker üzerinden eventlarla haberleşiyorlar. Böylece bir mikroservis A'nın tek işi bir event oluştuğunu bildirmek ve geri kalan işine devam etmek oluyor. B servisi paralelde message brokerdan aldığı datayı işleyip kendine ait işlemleri ilerletir.

Hibrit iletişim



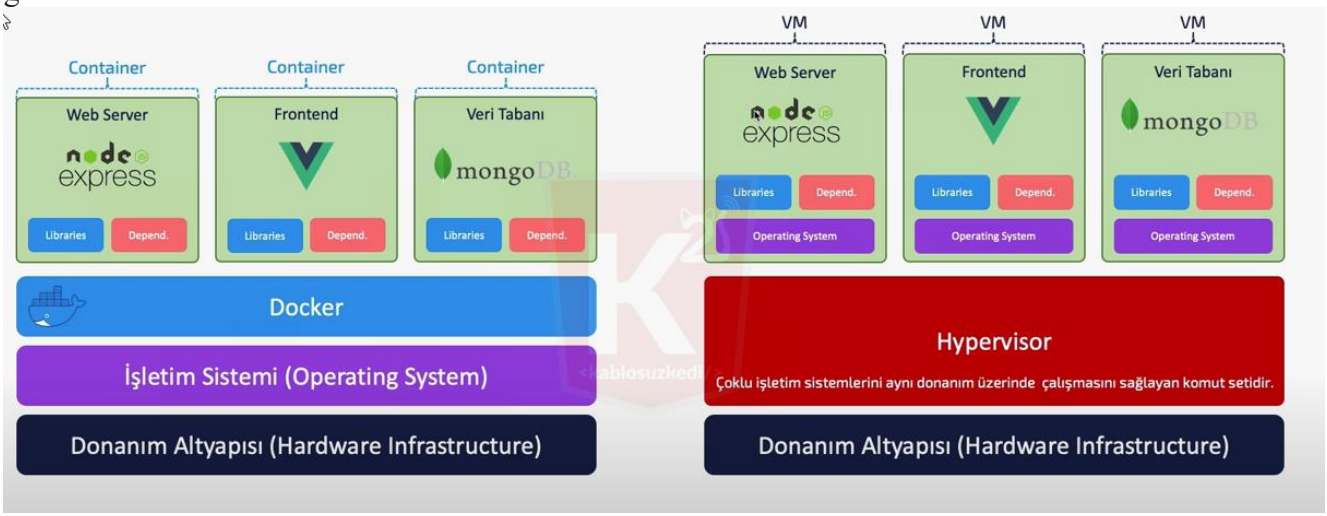
Hibrit iletişimde mikroservisler arası iletişim yaparken hem senkron hem de asenkron yapıyı kullanılır. A , B'ye senkron bir iletişim yaparken B ve C servisleri arasındaki iletişim asenkron bir iletişim söz konusudur.

11 - What do you mean by Domain driven design ?

Domain driven design(DDD) klasik tanımıyla kompleks business problemlerimizi efektif bir şekilde modellememize yarayan insan ilişkileri, mimari yaklaşım ve bu yaklaşımı uygularken izlememiz gereken prensipleri bizlere sunan domain centric bir yapıdır. DDD'ye göre yazılım projelerinde çıkan en büyük sorunlardan biri iletişim sorunudur. İş birimi ve teknik ekibin farklı bir dil konuşması, ihtiyaçların belirlenmesinde, geliştirilmesinde ve kapsamın büyütülmesinde uzun vadede sıkıntılar çıkarır ve ana hedef yani kod ile gerçeklenmesi gereken business probleminin bir anda teknoloji problemine dönüşmesi halini alır. DDD ise prensiplerini takip ederek iş birimi-teknik ekip arasındaki iletişim verimini maksimuma çıkarmayı ve bu süreci mimari anlamda çok iyi modellemeyi vaad eder.

12 – What is container in Microservices ?

Konteyner mimarisi de aslında bir sanallaştırma teknolojisidir, yalnızca hypervisor katmanı yerinde host işletim sistemi ve bu işletim sisteminin üzerinde konteyner motoru bulunur. Özetle aynı işletim sistemi üzerinde farklı uygulama veya servisleri sanallaştırarak çalışır. Aşağıdaki şekil ile docker mimarisi ve geleneksel sunucu sanallaştırma arasındaki farkı görebilirsiniz.

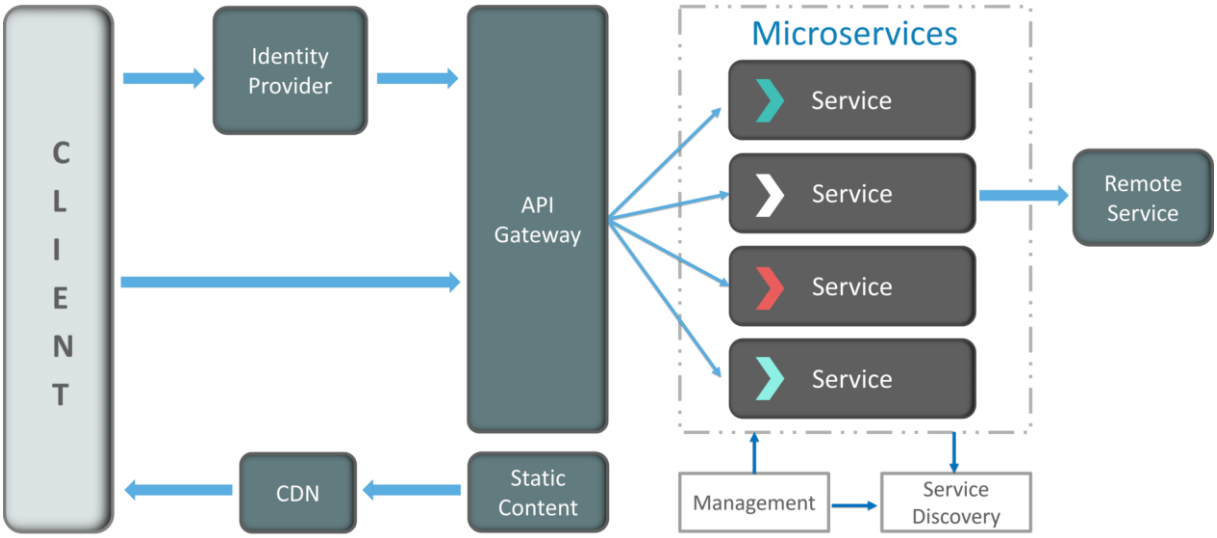


Günümüzde en popüler konteyner mimarisi olarak kullanılan Docker olsa da aslında bu işin atası 2008 yılında ilk sürümü çıkan Linux Containersdır. Yani aslında hypervisor tabanlı sanallaştırma kadar eskiye dayanan LXC geliştirilmiş, manuel olarak yapılan bir çok işlem ustaca otomatikleştirilmiş ve kullanım kolaylığının artmasıyla yaygınlığı da artmıştır.

Bazı yazılarda konteyner mimarisi daha iyidir çünkü hypervisor tabanlı sanallaştırmada işletim sistemine verdiğiniz kaynağın kullanılmayan kısmı çöp olur, bu yüzden verimlilik düşer gibi cümleler gördüm. Bu doğru bir yaklaşım değil, hypervisor tabanlı sanallaştırmada da atanan kaynakların kullanılmayan kısmı diğer sanal makineler tarafından kullanılabilir.

Burada sadece her sanal makine için ayrıca kurulan işletim sistemlerinin kapladığı disk alanı ve işletim sisteminin boştaki veya ayakta kalabilmesi için harcanan cpu ve memory kaynaklarının verimsizliği konuşmak doğru olabilir.

13 - What are the main components of Microservices architecture ?



Client

Mikroservis mimarisi, farklı istemci türleriyle başlar, farklı aygıtlar client olarak nitelendirilebilir. Client kullanıcının gördüğü arayüzlerle server side iletişimini sağlar. Aynı yapıda farklı tip clientlar mevcut olabilir. Tarayıcılar üzerinde çalışan web kısmı, mobilar kısımda çalışan android ve ios gibi nitelendirilebilir. Clientların asıl önemsediği kısım backend'e istekte bulunarak istenilen verinin kullanıcılar tarafından yönetilebilmesini sağlamaktır.

API Gateway

Clientlar mikroservislere doğrudan erişmez. API Gateway, clientların isteklerini uygun mikroservislere iletmek için bir giriş noktası görevi görür.

Clientlardan gelen tüm istekler önce API Gateway üzerinden geçer. Daha sonra istekleri uygun mikroservislere yönlendirir. HTTP ve WebSocket gibi web protokollerini içeride kullanılan farklı protokollere de çevirebilir. Amaca göre her farklı tipteki client'a özel API oluşturabilir.

Servisler arası iletişim(Rest, Message Bus)

Mikroservislerin birbirleri arasında iletişim kurdukları iki tür mesajlaşma yöntemi vardır:

Eşzamanlı Mesajlar: Clientlar bir isteğin yanıtını beklediği durumda, mikroservisler genellikle HTTP protokolünü kullanarak Rest haberleşirler.

Eşzamansız Mesajlar: Clientların bir servisten yanıt beklememesi durumunda ya da bir mikroservisde yapılan işlem hakkında, diğer mikroservisleri bilgilendirmek amacıyla; mikroservisler genellikle AMQP, STOMP, MQTT gibi protokolleri kullanarak message buslar aracılığı ile tek yönlü iletişim kurabilirler.

Veritabanları

Mikroservis, verilerini yakalamak ve ilgili iş işlevselliğini uygulamak için özel bir veritabanına sahiptir. Mikroservis veritabanları, service API'leri aracılığıyla güncellenir. Mikroservisler tarafından sağlanan servisler, farklı teknoloji yığınları için süreçler arası iletişimi destekleyen herhangi bir uzak servise taşınır.

Service Discovery

Bir mikroservis uygulamasında, çalışan servis örnekleri kümesi dinamik olarak değişir. Bulut sunucuları dinamik olarak atanmış ağ konumlarına sahiptir. Sonuç olarak, bir istemcinin servis talebinde bulunması için bir servis bulma mekanizması kullanması gerekir. Servis keşfinin önemli bir parçası servis kayıt defteridir.

CDN

Mikroservisler kendi içlerinde iletişim kurduktan sonra, statik içeriği, onları **Content Delivery Networks (CDN'ler)** aracılığıyla doğrudan istemcilere teslim edebilen bulut tabanlı bir depolama servisine dağıtırlar .

14 - How does a Microservice architecture work?

Mikroservis mimarisi; farklı cihazlardan gelen farklı istekleri (arama, oluşturma, yapılandırma ve diğer işlemler) farklı servisler üzerinden yönetmeye odaklanır.

Her bir modül, kullanım amacı ve işlevselliklerine göre ayrılır ve ayrı birer mikroservis haline getirilir. Bu mikroservisler, fonksiyonlarını yerine getirmek için kendi load-balance ve uygulama ortamlarına sahiptir ve aynı zamanda kendi veritabanlarında veri saklayabilirler.

Tüm mikroservisler birbiriyle stateless olarak REST ya da bir MessageBus üzerinden iletişim kurar. Mikroservisler, **Service Discovery** sayesinde iletişim kuracakları servislerin iletişim bilgilerine ulaşabilir ve servislerin durumlarını da izleyebilirler.

Clientlar tarafından oluşturulan tüm istekler API Gateway aracılığıyla mikroservislere iletilir. Böylece mikroservis kendi görevlerini yerine getirerek gerekli cevabı geri dönebilirler.