1) Senkron ve Asenkron iletişim nedir örneklerle açıklayın?

Senkron İletişim:

Senkron iletişimde, gönderici ve alıcı senkronizedir. Bu, göndericinin bir istek gönderdiği ve herhangi bir başka işlem yapmadan önce bir yanıt alıncaya kadar beklediği anlamına gelir.

Senkron iletişim genellikle engelleme (blocking) özelliğine sahiptir, yani gönderici bir yanıt alıncaya kadar engellenir.

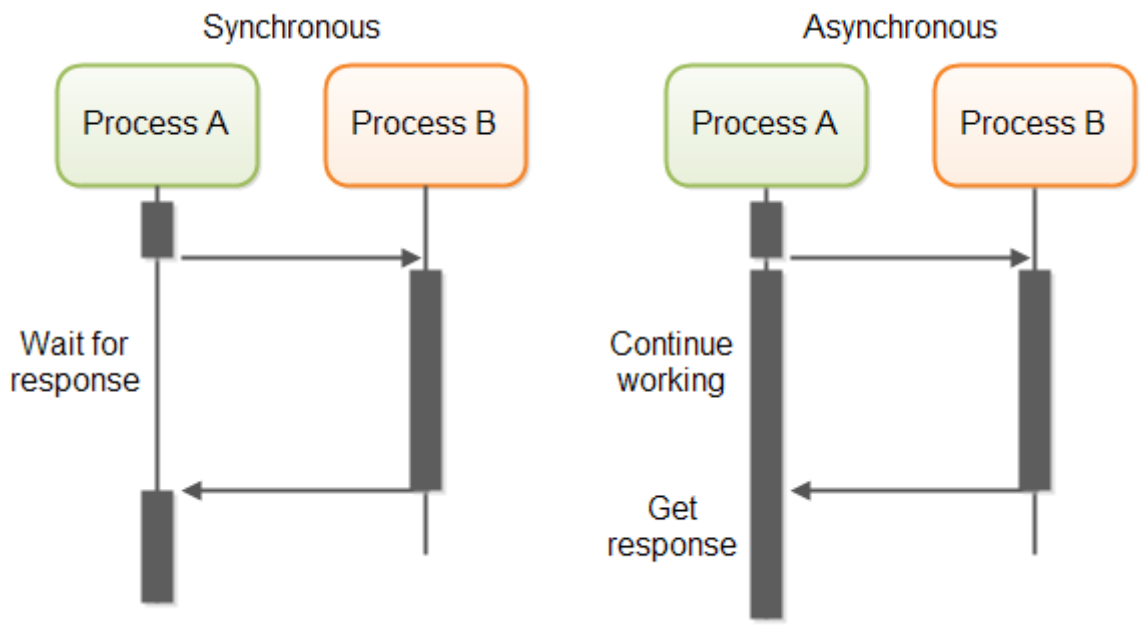
Örnek: Web uygulamalarındaki HTTP request-respond örnek verilebilir. Bir web sunucusuna (örneğin, bir tarayıcı kullanarak) bir istek gönderildiğinde, tarayıcı isteği işlenmesini ve bir yanıtın geri gönderilmesini bekler. Bu süre boyunca, tarayıcının başka bir şey yapmasına izin verilmez.

Asenkron İletişim:

Asenkron iletişimde, gönderici bir istek gönderir ancak hemen bir yanıt beklememektedir. Bunun yerine, diğer görevler veya aktivitelerle devam eder.

Asenkron iletişim, göndericiyi engellemez; gönderici, bir yanıt beklerken diğer görevleri gerçekleştirebilir.

Örnek: RabbitMQ veya Kafka gibi asenkron mesajlaşma sistemleri örnek verilebilir. Bu sistemlerde, bir mesaj üreticisi, bir tüketici tarafından hemen işlenmeyi beklemeksizin bir iletiyi bir kuyruğa veya konuya gönderir. Tüketici sonunda mesajı işleyecektir, ancak gönderici hemen yanıtı almayı beklemez.



2) RabbitMQ ve Kafka arasındaki farkları araştırın?

Kafka ve RabbitMQ, akış işlemede kullanılan mesaj kuyruğu sistemleridir. Aralarındaki farklar şöyle sıralanabilir:

İletişim Modeli:

RabbitMQ: RabbitMQ geleneksel bir mesaj kuyruğu modelini takip eder, burada mesajlar üreticiler tarafından alışverişlere gönderilir, ardından kuyruklara yönlendirilir ve nihayetinde tüketiciler tarafından tüketilir. Noktadan-noktaya, yayımla-abone ol ve istek-cevap gibi çeşitli mesajlaşma desenlerini destekler.

Kafka: Kafka dağıtılmış bir taahhüt kayıt mimarisini takip eder. Öncelikle dağıtılmış bir olay akışı platformu olarak tasarlanmıştır. Veriler konulara düzenlenir ve mesajlar bu konuların içindeki bölümlere eklenir. Tüketiciler konulara abone olur ve bölümlerden mesajları okur. Kafka, mesajları belirli bir süre saklar, bu da gerçek zamanlı veri işleme ve analiz için uygun hale getirir.

Ölçeklenebilirlik ve Performans:

RabbitMQ: RabbitMQ daha geleneksel bir mesaj aracıdır ve mesaj sıralaması ve işlem güvenilirliği önemli olduğunda uygun bir seçenektir. Ancak, yüksek veri işleme ve büyük ölçekli veri işleme durumlarında ölçeklenebilirlik zorluklarıyla karşılaşabilir.

Kafka: Kafka oldukça ölçeklenebilir ve büyük veri hacimlerini ve yüksek verim yüklerini işlemek için tasarlanmıştır. Dağıtılmış mimarisi, birden fazla düğme üzerinde yatay ölçeklenmeyi mümkün kılar. Kafka'nın performansı, milyonlarca mesajı saniyede işleyebilme yeteneği ile özellikle dikkat çekicidir.

Dayanıklılık ve Kalıcılık:

RabbitMQ: RabbitMQ, mesaj dayanıklılığını kalıcı kuyruklar aracılığıyla sağlar, böylece sistem hatalarında bile mesajların kaybolmadığından emin olunur. Mesaj kalıcılığı için farklı depolama backend'lerini destekler, bunlar arasında disk ve bellek bulunur.

Kafka: Kafka yüksek dayanıklılık ve hata toleransı için tasarlanmıştır. Mesajları diske kaydeder ve bunları veri erişilebilirliği ve güvenilirliği sağlamak için birden çok aracıya replike eder. Kafka'nın dayanıklılığı, veri bütünlüğünün kritik olduğu kullanım durumları için uygundur.

Mesaj Saklama Süresi:

RabbitMQ: RabbitMQ genellikle tüketiciler tarafından tüketildikten sonra mesajları saklamaz, bunu açıkça yapılandırılmamışsa. Daha çok gerçek zamanlı mesaj teslimi üzerine odaklanır.

Kafka: Kafka, mesajları yapılandırılabilir bir saklama süresi boyunca saklar, bu da tüketicilerin geçmiş verileri yeniden oynatabilmesine veya tüketmesine olanak tanır. Bu özellik, veri boru hatları, olay kaynağı ve akış işleme uygulamaları oluşturmak için Kafka'nın uygun olmasını sağlar.

Kullanım Durumları:

RabbitMQ: RabbitMQ, görev dağıtımı, iş kuyrukları ve mikroservisler arasında asenkron iletişim gibi geleneksel mesajlaşma senaryoları için uygun bir araçtır.

Kafka: Kafka genellikle gerçek zamanlı veri işleme, olay odaklı mimariler, günlük birleştirme, ölçümler toplama ve akış işleme ve analiz uygulamaları oluşturmak için kullanılır.

Performans:

Hem RabbitMQ hem de Kafka, kullanım amaçları için yüksek performanslı mesaj iletimi sunar. Ancak, Kafka, mesaj iletim kapasitesinde RabbitMQ"dan daha iyi performans gösterir.

Kafka, yüksek aktarım hızına sahip mesaj alışverişini etkinleştirmek için sıralı disk G/Ç kullandığı için saniyede milyonlarca mesaj gönderebilir. Sıralı disk G/Ç, bitişik bellek alanından veri depolayan ve erişen bir depolama sistemidir ve rastgele disk erişiminden daha hızlıdır.

RabbitMQ de saniyede milyonlarca mesaj gönderebilir, ancak bunu yapmak için birden fazla aracı gerektirir. Tipik olarak, RabbitMQ'nun performansı saniyede ortalama binlerce mesajdır ve RabbitMQ'nun kuyrukları sıkışık ise yavaşlayabilir.

Güvenlik:

RabbitMQ ve Kafka, uygulamaların güvenli bir şekilde ancak farklı teknolojilerle mesaj alışverişi yapmasına olanak tanır.

RabbitMQ, kullanıcı izinlerini ve aracı güvenliğini yönetmek için yönetim araçlarıyla birlikte gelir.

Bu arada, Apache Kafka mimarisi, TLS ve Java Kimlik Doğrulama ve Yetkilendirme Hizmeti (JAAS) ile güvenli etkinlik akışları sağlar. TLS, mesajlarda istenmeyen gizlice dinlemeyi önleyen bir şifreleme teknolojisidir ve JAAS hangi uygulamanın aracı sistemine erişimi olduğunu kontrol eder.

Mimari yaklaşım:

Bir RabbitMQ aracısı, aşağıdaki bileşenlerle düşük gecikme süresi ve karmaşık mesaj dağıtımlarına olanak tanır:

Alışveriş, üreticiden mesajlar alır ve nereye yönlendirilmesi gerektiğini belirler

Kuyruk, bir aracıdan mesajlar alan ve bunları tüketicilere gönderen depolamadır

Bağlama, bir alışverişi ve bir aracıyı birbirine bağlayan bir yoldur

RabbitMQ'da yönlendirme anahtarı, mesajları bir alışverişten belirli bir kuyruğa yönlendirmek için kullanılan bir mesaj öz niteliğidir. Bir üretici bir alışverişe mesaj gönderdiğinde, mesajın bir parçası olarak bir yönlendirme anahtarı içerir. Alışveriş daha sonra mesajın hangi kuyruğa teslim edilmesi gerektiğini belirlemek için bu yönlendirme anahtarını kullanır.

Kafka kümesi, daha karmaşık bir mimariyle yüksek aktarım hızına sahip akış etkinliği işleme sağlar. Bunlar bazı önemli Kafka bileşenleridir:

Kafka aracısı, üreticilerin tüketicilere veri akışı yapmasına olanak tanıyan bir Kafka sunucusudur. Kafka aracısı konuları ve ilgili bölümlerini içerir.

Konu, bir Kafka aracısındaki benzer verileri gruplandıran veri depolamadır.

Bölüm, tüketicilerin abone olduğu bir konu içinde daha küçük veri depolamadır.

ZooKeeper hataya dayanıklı akış sağlamak için Kafka kümeleri ve bölümlerini yöneten özel bir yazılımdır.

Kafka'daki üreticiler her mesaj için bir mesaj anahtarı atar. Ardından, Kafka aracısı mesajı söz konusu konunun önde gelen bölümünde saklar.

3) Docker ve Virtual Machine nedir?

Docker:

Docker, uygulamaları geliştirmek, dağıtmak ve çalıştırmak için kullanılan bir platformdur. Geliştiricilere bir uygulamayı ve bağımlılıklarını bir standart birim olan bir konteynıra paketleme imkanı sağlar. Konteynerler hafif, taşınabilir ve kendi kendine yeterlidir; kodu, çalışma zamanını, kütüphaneleri ve bağımlılıkları içeren her şeyi çalıştırmak için gereken her şeyi kapsarlar.

Docker'ın temel özellikleri şunlardır:

Konteynerleştirme: Docker konteynerleri, farklı ortamlarda çalıştırıldıklarında bile uygulamaların tutarlı bir şekilde çalışmasını sağlayan izolasyon ve tutarlılık sağlar.

Taşınabilirlik: Docker konteynerleri, Docker'ı destekleyen herhangi bir altyapıda çalıştırılabilir; bu da uygulamaların ortamlar arasında kolayca taşınmasını sağlar.

Kaynak Verimliliği: Docker konteynerleri, geleneksel sanal makinelerden daha az kaynak kullanarak çalışır, çünkü konteynerler ana işletim sisteminin çekirdeğini paylaşırlar.

Ölçeklenebilirlik: Docker, uygulamaları iş yükü taleplerini karşılamak için hızla ek konteynerler dağıtarak kolayca ölçeklendirebilir.

Sürüm Kontrolü: Docker imajları, konteynerler oluşturmak için kullanılır ve sürümü alınır; bu da geliştiricilerin değişiklikleri izlemesine ve ihtiyaç duyulursa önceki sürümlere geri dönmesine olanak tanır.

Sanal Makine (VM):

Sanal makine, fiziksel bir bilgisayar sisteminin emülasyonudur ve bir fiziksel ana makinede aynı anda birden fazla sanallaştırılmış işletim sisteminin çalışmasına olanak tanır; her biri kendi CPU'su, belleği, depolaması ve ağ arabirimi ile.

Sanal makinelerin temel bileşenleri şunlardır:

Hypervisor: Hypervisor veya sanal makine izleyicisi (VMM) olarak bilinen, fiziksel ana makinede sanal makineleri oluşturan ve yöneten yazılım veya firmware'dir. Hardware kaynaklarını sanal makineler arasında tahsis eder ve izole eder.

Konuk İşletim Sistemleri: Sanal makineler konuk işletim sistemlerini çalıştırır; bu, ana işletim sisteminden farklı olabilir. Her sanal makine kendi işletim sistemi örneğine sahiptir ve aynı fiziksel ana makinedeki diğer sanal makinelerden bağımsız olarak çalışır.

Sanal Donanım: Sanal makinelerin sanallaştırılmış donanım bileşenleri vardır; bunlar sanal CPU'lar, sanal bellek, sanal diskler ve sanal ağ arabirimlerini içerir. Bu sanal donanım bileşenleri, hypervisor tarafından yönetilir ve konuk işletim sistemlerinin ve uygulamalarının çalışması için gerekli kaynakları sağlar.

İzolasyon: Sanal makineler farklı işletim sistemleri ve uygulamalar arasında güçlü izolasyon sağlar. Bu izolasyon, aynı fiziksel ana makinedeki diğer sanal makineler üzerindeki hataların veya güvenlik ihlallerinin etkileşmesini önler.

Esneklik: Sanal makineler kolayca oluşturulabilir, değiştirilebilir ve silinebilir; bu da yazılım uygulamalarının esnek dağıtımı ve yönetimini sağlar.

Dolayısıyla sanal makineler, tek bir fiziksel makinede birden çok işletim sistemi ve uygulamanın izole örneklerinin çalıştırılmasını sağlayarak fiziksel donanım kaynaklarının verimli kullanılmasını sağlar.

4) Docker ile RabbitMQ ve PostgreSQL ve ya MySQL kurulumu yapın?

5) Docker komutlarını örneklerle açıklayın.

1- docker images

Bu komut, tüm Docker imajlarını ve bunların deposunu, etiketini ve boyutunu listeler.

2- docker rm my_container

Bu komut, "my_container" adında bir konteyneri kaldırır.

3- docker ps

Bu komut, çalışan tüm konteynerleri ve detaylarını (konteyner ID'si, imaj, durum ve adlar gibi) listeler.

4- docker run

Bu komut, bir Docker imajından bir konteyner oluşturup başlatmak için kullanılır.

Örneğin “docker run -d --name my_container nginx” komutu "nginx" imajını kullanarak "my_container" adında bir konteyner oluşturur ve başlatır.

5- docker pull nginx

Bu komut, Docker Hub deposundan "nginx" imajının en son sürümünü çeker.

6- docker exec

Bu komut, çalışan bir konteynerde komut çalıştırmak için kullanılır.

Örneğin “docker exec -it my_container bash” komutu ile "my_container" adındaki çalışan konteynerde etkileşimli olarak bir bash kabuğu başlatır.

7- docker build

Bu komut, bir Docker dosyasından bir Docker imajı oluşturmak için kullanılır.

Örneğin “docker build -t my_image .” komutu ile mevcut dizindeki Docker dosyasını kullanarak "my_image" adında bir Docker imajı oluşturulur.

8- docker pull

Pull komutu ile istenen image Docker Hub’dan kullanılan makineye çekilebilir.

9- docker container stop

Bu komut ile bir container durdurulabilir.

6) Microservice ve Monotlith mimarilerini kıyaslayın.

Monolithic yaklaşım, bir sistemin/nesnenin/olgunun tek bir parça olacak şekilde tasarlanmasıdır. Monolithic mimari ise bu tasarımın stratejik yapılanmasıdır. Monolithic yaklaşım, üretilecek sistemin/nesnenin/olgunun bileşenlerini(component) birbirlerine bağlı(interdependent) olarak ve kendi kendine yetecek(self-contained) şekilde tasarlanmasını sağlayan ve böylece tek bir bütünsel varlık olarak nihai sonuca varılmasını sağlayan mimaridir.

Microservice mimari, birbirinden bağımsız olarak çalışan ve birbirleriyle haberleşerek bir bütün olarak hareket eden servis(ler) yapılanmasıdır. Her servisin bir diğerinden bağımsız olarak iş mantığını yürütmesi ve bir başka servis ile ilgilenmemesi, bir onarım yahut restorasyon durumunda uygulamanın bütününü etkilemeyeceğinden dolayı sürdürülebilirlik ilkesi desteklenmiş olacak ve böylece bodoslama olarak tabir edeceğimiz monolithic yaklaşımındaki karmaşıklığı ortadan kaldırmış ve yönetimi daha da kolaylaştırmış olacaktır.

Monolithic mimarisinin avantajları:

-Yönetilebilirlik ve monitoring kolaydır.

-Küçük çaplı projeler için geliştirilmesi ve bakımı kolaydır. Hızlı bir şekilde uygulama geliştirilebilir.

-Fonksiyonalitelerin birlikte çalışabilirliği tutarlıdır.

-Transaction yönetimi kolaydır.

Monolithic mimarisinin dezavantajları:

-Uygulama büyüdükçe yeni özellik geliştirilmesi ve mevcut kodun bakımı zorlaşır.

-Projede çalışan ekip sayısının ve çalışan sayısının artması ile geliştirme ve bakım daha güç hale gelir.

-Birbirlerine olan bağımlılıklarından dolayı, bir fonksiyonalitede yapılan değişiklik diğer yerleri etkileyebilir.

-Spesifik bir fonksiyonaliteyi scale edebilme imkanı yoktur. (Örneğin geliştirdiğiniz uygulamada sürekli fatura oluşturuluyor ve burası uygulamanın dar boğazı. Siz bu fonksiyonaliteyi birden fazla instance da çalıştırmak isteseniz bile uygulamanız monolithic mimaride olduğu için sadece ilgili servis yerine bütün uygulamayı scale etmek zorunda kalırsınız.)

-Versiyon yönetimi zorlaşır.

-Uygulamada aynı programlama dili ve aynı frameworklerin kullanılması gerekir.

-Uygulamada yapılan küçük bir değişiklikte bile bütün uygulamanın deploy olması gerekir.

Microservice mimarisinin avantajları:

-Uygulama çok büyük de olsa çok küçük de olsa yeni özellik eklenmesi ve mevcut kodun bakımı kolaydır. Çünkü sadece ilgili servis içerisinde değişiklik yapmak yeterlidir.

-Her servis birbirinden bağımsız ve sadece kendi iş mantıklarını bulundurduğu için servisin code base'i oldukça sade olacaktır.

-Ekipler daha verimli ve hızlı bir şekilde çalışabilir.

-Servisler birbirinden bağımsız bir şekilde scale edilebilir.

-Versiyonlama kolay bir şekilde gerçekleştirilir.

-Bir servisde yapılan değişiklik, diğer servislerin deploy yapılmasını gerektirmez. Sadece ilgili servisin deploy yapılması yeterlidir.

-Servisler farklı dillerde ve farklı frameworkler ile yazılabilir. Her servisin kendine ait farklı veritabanı olabilir.

Dezavantajları:

-Birden fazla servis ve birden fazla veritabanı olduğu için transaction yönetimi zorlaşacaktır.

-Bu servislerin yönetilebilirliği ve monitoring işlemi zorlaşacaktır.

-Servisler gereğinden fazla bağımsız olursa yönetimi zorlaşacaktır. Örneğin user ile alaklı CRUD işlemler yapan user-service adında bir servisiniz olduğunu varsayalım. Eğer user-service'ın her bir fonksiyonalitesini (create, update, delete, get... vb) ayrı servislere ayırmak isterseniz bunun yönetimi çok daha zorlaşacaktır.

7) API Gateway, Service Discovery, Load Balancer kavramlarını açıklayın.

API Gateway:

Bir API Gateway, mikroservis mimarilerinde ve modern web geliştirmede önemli bir bileşendir. İstemcilerin çeşitli arka uç servicelerine ve API'lere erişmek için tek bir giriş noktası görevi görür. İstemci erişimini basitleştirir, güvenliği artırır ve trafik yönetimi, güvenlik uygulamaları ve izleme gibi çeşitli ek özellikler sunar.

API Gateway, istemciler için tek bir giriş noktası olarak hizmet eder ve altındaki mikroservis mimarisinin karmaşıklığını gizler. İstemcilerin doğrudan bireysel serviceleri çağırması yerine, istekleri uygun servicelere yönlendirir.

API Gateway'in temel işlevlerinden biri, gelen istekleri uygun arka uç servicelerine yönlendirmektir. Gelen istekleri URL yollarına veya başlıklara göre analiz eder ve bunları ilgili service'ye ileterek yönlendirir. Bu yönlendirme mantığı, istek yöntemine, yola, başlıklara veya sorgu parametrelerine dayanabilir.

Bazı durumlarda, bir istemcinin isteği birden fazla backend servicesinden veri gerektirebilir. API Gateway, bu istekleri birleştirir, farklı servicelerden gerekli verileri toplar ve sonuçları tek bir yanıtta birleştirir. Bu, istemci ile backend serviceleri arasındaki dönüş sayısını azaltarak verimliliği ve performansı artırır.

API Gateway, farklı protokoller ve veri formatları arasında çeviri yapabilir. Örneğin, HTTP/1.1 veya HTTP/2 üzerinden gelen istekleri kabul edebilir ve HTTP, gRPC veya WebSockets gibi farklı protokoller kullanarak backend serviceleriyle iletişim kurabilir. Ayrıca, JSON, XML veya Protokol Buffers gibi formatlar arasında veri dönüşümü yapabilir.

API Gateway genellikle backend servicelerinin yerine kimlik doğrulama ve yetkilendirme işlemlerini yönetir. Güvenlik politikalarını uygulayabilir, kullanıcı kimlik bilgilerini doğrulayabilir ve erişim belgeleri oluşturabilir. Bu merkezi kimlik doğrulama mekanizması, güvenlik uygulamalarını basitleştirir ve bireysel serviceler üzerindeki yükü azaltır.

Backend servicelerini kötüye kullanımdan veya aşırı yüklenmeden korumak için API Gateway, oran sınırlama ve kısıtlama politikalarını uygulayabilir. Belirli bir zaman dilimi içinde bir istemcinin yapabileceği istek sayısını sınırlayabilir veya belirli eşiklere dayalı olarak istek oranını sınırlayabilir. Bu, sistem istikrarını korumaya ve service reddi saldırılarını önlemeye yardımcı olur.

API Gateway genellikle gelen istekleri, yanıtları ve meta verileri izleme, hata ayıklama ve denetim amaçlarıyla kaydeder. Sistemin içinden geçen trafiği görselleştirir, işletmelere performans sorunlarını tanımlama, kullanım desenlerini izleme ve hataları etkili bir şekilde giderme olanağı sağlar.

Service Discovery:

Service Discovery, mikroservis mimarilerinde temel bir kavramdır. Bir ağ içinde mevcut olan service'leri dinamik olarak keşfetme ve bulma sürecini ifade eder. Service Discovery basitleştirir, hata toleransını artırır ve servicelerin sürekli olarak dağıtıldığı, ölçeklendirildiği ve güncellendiği dinamik altyapı ortamlarını destekler.

Dynamic Service Registration:

Service'lerin sık sık dağıtıldığı, ölçeklendirildiği veya sonlandırıldığı dinamik bir ortamda, Service Discovery, service'lerin kendilerini bir merkezi kayıt defteri veya Service Registry'e dinamik olarak kaydetmelerine olanak tanır. Bu kayıt süreci, servicenin ağdaki konumu, endpoint adresleri ve meta veriler gibi bilgileri içerir.

Service Registry:

Service Registry, mevcut servicelerin bir listesini ve ağdaki konumlarını koruyan merkezi bir veritabanı veya dizindir. Service meta verileri için gerçek bir kaynak olarak hareket eder ve istemcilerin serviceleri dinamik olarak sorgulamaları için bir kaynak olarak görev yapar. Service Registry örnekleri arasında Consul, etcd, ZooKeeper ve Kubernetes'in service discovery mekanizmaları bulunur.

Service Lookup:

İstemciler, servicelerin ağ adreslerini veya yapılandırmalarını sert kodlamak zorunda kalmadan dinamik olarak serviceleri aramak ve bulmak için Service Registry'i kullanır. Servicelerin kesin IP adreslerini veya uç noktalarını bilmek yerine, istemciler service'nin gereken bilgilerini almak için Service Registry'i sorgularlar, örneğin, ana bilgisayar, bağlantı noktası ve protokol.

Load Balancing:

Service Discovery genellikle Load Balancing ile birlikte kullanılır. Bir istemci birden fazla service instancesini keşfettiğinde, gelen istekleri bu örnekler arasında eşit olarak dağıtmak için Load Balancing algoritmalarını kullanabilir. Bu, iş yükünü birden çok service instancesi arasında eşit bir şekilde dağıtarak, hata toleransını, ölçeklenebilirliği ve performansı artırır.

Health Checking:

Service Discovery sistemleri genellikle servicelerin durumunu ve erişilebilirliğini izlemek için sağlık kontrol mekanizmaları içerir. Serviceler, sağlık durumlarını belirtmek için belirli aralıklarla heartbeat sinyalleri veya sağlık kontrol mesajları gönderirler. Bir service kullanılamaz hale gelirse veya sağlıksız hale gelirse, Service Registry, meta verilerini buna göre güncelleyerek istemcilerin problemli örneklerden kaçınmasını sağlayabilir.

Service Auto-scaling:

Service Discovery, talebe dayalı olarak servicelerin dinamik olarak ölçeklendirildiği otomatik ölçeklendirme ortamlarında önemli bir rol oynar. Bir servicenin yeni instanceleri dinamik olarak eklenir veya kaldırılırsa, bunlar kendilerini Service Registry'e kaydeder veya kayıtlarını siler, böylece istemcilerin en son mevcut örnekleri keşfetmesi ve bunlarla etkileşime girmesi sağlanır.

Decentralized Service Discovery:

Centralized Service Registryler yaygın olsa da, bazı mimariler serviceleri doğrudan birbirleriyle iletişim kurduğu veya Service Discovery için eşten eşe protokoller kullandığı merkezi olmayan yaklaşımları tercih eder. Merkezi Olmayan Service Discovery, daha düşük gecikme süresi, azaltılmış ağ üzerindeki yük ve geliştirilmiş hata toleransı sağlayabilir, ancak ek karmaşıklık getirebilir.

Load Balancer:

Load Balancer gelen ağ trafiğini birden çok backend sunucusu veya instance'ı arasında eşit bir şekilde dağıtarak kaynakların optimal kullanımını, yüksek erişilebilirliği ve ölçeklenebilirliği sağlar. Load Balancer, gelen istekleri veya ağ trafiğini birden çok backend sunucusu veya instance'ı arasında dağıtır. Bu dağıtım, Round Robin, Least Connections veya IP Hashing gibi önceden tanımlanmış algoritmalar temelinde gerçekleşir. Yükü dengeli bir şekilde dağıtarak, herhangi bir sunucunun aşırı yük altında kalmasını önler ve böylece genel sistem performansını ve tepki süresini artırır.

Load Balancer'lar, sistem güvenilirliğini ve erişilebilirliğini artırarak failover yetenekleri sağlar. Bir sunucu veya instance donanım arızası, bakımı veya aşırı yüklenmesi nedeniyle kullanılamaz hale geldiğinde, Load Balancer trafiği diğer sağlıklı sunuculara yönlendirir, böylece kesintisiz hizmet sağlanır. Bu yedekleme, kesinti süresini en aza indirir ve sistemin genel dayanıklılığını artırır.

Load Balancer'lar, yatay ölçeklenebilirliği destekleyerek sistemlerin gerektiğinde daha fazla sunucu veya instance eklemesine olanak tanır. Hizmetlere olan talep arttıkça, ek sunucular sağlanabilir ve otomatik olarak Load Balancer'ın döngüsüne entegre edilebilir. Benzer şekilde, talep azaldığında sunucular, işlemleri etkilemeden kaldırılabilir. Bu ölçeklenebilirlik, sistemlerin değişen trafik seviyelerini yönetmesini ve artan kullanıcı yüklerini etkili bir şekilde karşılamasını sağlar.

Sağlık kontrol mekanizmalarını kullanarak backend sunucularının veya instance'ların sağlık ve durumunu sürekli olarak izler. Bu sağlık kontrolleri, her sunucunun kullanılabilirliğini ve yanıt verme durumunu düzenli olarak kontrol eder. Bir sunucu yanıt vermezse veya yanıt vermez hale gelirse, Load Balancer otomatik olarak onu dönüşten çıkarır ve sadece sağlıklı sunucuların gelen trafiği işlemesini sağlar.

Belirli senaryolarda, oturum bağlılığını veya sürekliliğini sağlamak gerekebilir; bu, aynı istemcinin ardışık isteklerinin aynı backend sunucusuna veya instance'ına yönlendirilmesini sağlar. Load Balancer'lar, oturum sürekliliğini destekleyerek bir istemcinin oturumunu belirli bir sunucuyla ilişkilendirir. Bu, devamlılığı ve sürekliliği, sürekli etkileşim gerektiren durumlu uygulamalar veya oturumlar için sağlar.

Load Balancer'lar, SSL/TLS şifreleme ve şifre çözme görevlerini backend sunuculardan uzaklaştırarak, SSL sonlandırma olarak bilinen bir süreçle performansı artırır. SSL bağlantılarını Load Balancer'lar da sonlandırarak, backend sunuculardaki hesaplama yükünü azaltabilir, performansı artırabilir ve sertifika yönetimini basitleştirebilir. Ayrıca, Load Balancer'lar, istemci ve Load Balancer'lar arasındaki trafiği şifreleyerek güvenliği ve gizliliği artırabilir.

Birden çok bölge veya veri merkezi boyunca dağıtılmış uygulamalar için, Global Load Balancer'lar, jeo-konumsal yakınlık, ağ koşulları veya gecikme gibi kriterlere dayalı olarak trafiği en yakın veya en uygun uca yönlendirebilir. Bu, kullanıcı deneyimini geliştirir, gecikmeyi en aza indirir ve kullanıcının konumundan bağımsız olarak veri aktarımını maksimize eder.

8) Hibernate, JPA, Spring Data framework'lerini örneklerle açıklayın.

Hibernate:

Hibernate Java uygulamaları için bir nesne ilişkisel haritalama (ORM) framework'üdür. Database interaction ve query management için çeşitli özellikler sağlamanın yanı sıra, Java nesnelerini database tablolarına veya bunun tersini yapmayı basitleştirir. ORM ise nesne odaklı dillerdeki nesnelerin, ilişkisel veri tabanlarındaki kayıtlara nasıl karşılık geldiğini yürüten bir teknolojidir.

Kod örneği:

```
// Entity databasedeki bir tabloyu temsil eder

@Entity

@Table(name = "books")

public class Book {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;


    @Column(name = "title")
```

```
    private String title;

    @Column(name = "author")
    private String author;

    // Getters and setters
}
// database operasyonları yapmak için hibernate sessionları kullanılıyor
Session session = sessionFactory.openSession();

// Create
Book book = new Book();
book.setTitle("Harry Potter");
book.setAuthor("J.K. Rowling");
session.beginTransaction();
session.save(book);
session.getTransaction().commit();

// Read
Book retrievedBook = session.get(Book.class, 1L);
System.out.println("Title: " + retrievedBook.getTitle() + ", Author: " +
retrievedBook.getAuthor());

// Update
retrievedBook.setTitle("New Title");
session.beginTransaction();
session.update(retrievedBook);
session.getTransaction().commit();

// Delete
```

```
session.beginTransaction();

session.delete(retrievedBook);

session.getTransaction().commit();


session.close();
```

JPA:

Java Persistence API (JPA), Java platformunda nesne ilişkilendirme teknolojileri için bir standart belirleyen bir API'dir. JPA, Hibernate gibi ORM framework'lerinin temelini oluşturur ve Java uygulamalarının veritabanı işlemlerini yönetmek için bir arayüz sağlar. JPA, ORM'i standartlaştıran bir API'dir, Hibernate ise bu API'yi uygulayan ve genişleten bir Java ORM çerçevesidir.

Örnek kod:

```
// Entity databasedeki bir tabloyu temsil eder

@Entity

@Table(name = "books")

public class Book {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;


    @Column(name = "title")

    private String title;


    @Column(name = "author")

    private String author;


    // Getters and setters

}

// JPA database operationları için EntityManager kullanır

EntityManager entityManager = entityManagerFactory.createEntityManager();
```

```
EntityTransaction transaction = entityManager.getTransaction();

// Create
Book book = new Book();
book.setTitle("Harry Potter");
book.setAuthor("J.K. Rowling");
transaction.begin();
entityManager.persist(book);
transaction.commit();

// Read
Book retrievedBook = entityManager.find(Book.class, 1L);
System.out.println("Title: " + retrievedBook.getTitle() + ", Author: " +
retrievedBook.getAuthor());

// Update
retrievedBook.setTitle("New Title");
transaction.begin();
entityManager.merge(retrievedBook);
transaction.commit();

// Delete
transaction.begin();
entityManager.remove(retrievedBook);
transaction.commit();

entityManager.close();
```

Spring Data:

Spring Data, veritabanı etkileşimlerini kolaylaştırmak için Spring tarafından sağlanan bir proje ve bir dizi modül içerir. Spring Data, JPA veya Hibernate gibi ORM framework'lerini kullanarak CRUD (create, read, update, delete) işlemlerini gerçekleştirmeyi kolaylaştırır.

Örnek kod:

```
// Spring Data JPA Repository örneği
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
}
// Spring Data JPA kullanarak bir kullanıcı arama ve kaydetme örneği
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    @Transactional
    public void saveUser(User user) {
        userRepository.save(user);
    }

    public User findUserByUsername(String username) {
        return userRepository.findByUsername(username);
    }
}
```