1. Senkron ve Asenkron iletişim nedir örneklerle açıklayın?

   Bir mikroservis, diğer bir mikroservise request gönderiyor ve sonucunu bekliyorsa bu iletişim senkron bir mikroservis iletişimidir. Örneğin bir online ödeme işleminde, öncelikle ödeme servisine ödeme isteği gelir. Sonrasında ödeme servisi banka servisiyle iletişime geçerek ödeme yapılması beklenir. Ödeme yapıldığında banking servisi bir response üretir. Üretilen response'u hali hazırda cevabı bekleyen payment servisi alarak gerekli işlemleri kendi içinde gerçekleştirir ve kullanıcıya cevabını döner. Bu iletişim akışının gerçekleşmesinde servisler, bir diğer servisten sonuç beklemiştir ve uygun aksiyonu almıştır.

   Asenkron işlemlerde istekte bulunan servis, hizmet aldığı servisten yanıt beklemez. Böylece istekte bulunan servis hizmet aldığı serviste bağımlı hale gelmezler. Asenkron istek başarısız olduğunda bile servis sorunsuzca çalışmaya devam eder. Başarısız olma durumu ise farklı tekniklerle çözümlenir ama bu durum herhangi bir işlemin devam etmesine engel olmaz. Buna örnek olarak ise mesajlaşma uygulamaları verilebilir. Bir kullanıcının gönderdiği mesaj, alıcının herhangi bir davranışına bağlı değildir. Bağımsız işlemlerdir. [1]

2. RabbitMQ ve Kafka arasındaki farkları araştırın.

   Kafka ve RabbitMQ, veri akışının hızlı ve etkili bir şekilde işlenmesini sağlayan mesaj kuyruğu sistemleridir. Veri akışı, sürekli ve yüksek hacimli verileri içerir. Örneğin, sensör verileri gibi gerçek zamanlı değişen verilerin toplanması ve işlenmesi gerekebilir. RabbitMQ, farklı kaynaklardan gelen akış verilerini toplar ve işlemek üzere farklı hedeflere yönlendirir. Apache Kafka ise gerçek zamanlı veri hatları ve akış uygulamaları oluşturmak için kullanılan açık kaynaklı bir platformdur. Kafka, RabbitMQ'dan daha geniş yeteneklere sahip, yüksek ölçeklenebilirlik, hata toleransı ve dayanıklı mesajlaşma sistemleri sunar. [2]

	RabbitMQ	Kafka
Mimari	RabbitMQ'nun mimarisi karmaşık mesaj yönlendirme için tasarlanmıştır. İtme modelini kullanır. Üreticiler tüketicilere farklı kurallara sahip mesajlar gönderir.	Kafka, gerçek zamanlı, yüksek verimli akış işleme için bölüm tabanlı tasarım kullanır. Çekme modelini kullanır. Üreticiler, tüketicilerin abone olduğu konulara ve bölümlere mesajlar yayınlar.
İleti İşleme	RabbitMQ aracıları mesaj tüketimini izler. Tüketildikten sonra iletileri siler. Mesaj önceliklerini destekler.	Tüketiciler, bir ofset izleyici ile mesaj alımını takip eder. Kafka, mesajları saklama politikasına göre saklar. Mesaj önceliği yoktur.
Performans	RabbitMQ düşük gecikme süresine sahiptir. Saniyede binlerce mesaj gönderir.	Kafka saniyede milyonlarca mesaja kadar gerçek zamanlı iletim sağlar.
Programlama dili ve protokolü	RabbitMQ çok çeşitli dilleri ve eski protokolleri destekler.	RabbitMQ çok çeşitli dilleri ve eski protokolleri destekler.

3. Docker ve Virtual Machine nedir?

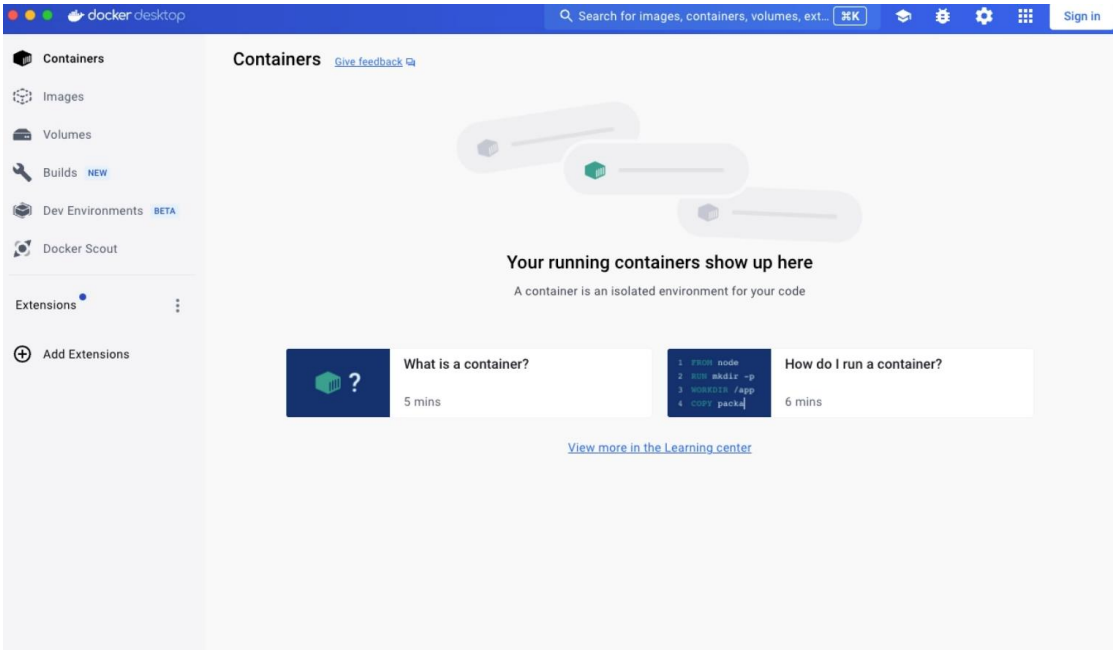
   Sanallaştırma, sunucular ve ağlar gibi gerçek hayattaki kaynakların sanal bir örneğini oluşturmaya olanak tanır. Docker ve Virtual Machine'ler sanallaştırma işlemini yaparlar.

   Sanallaştırma, bir fiziksel bilgisayar üzerinde birden fazla sanal makinenin oluşturulmasını sağlayan bir teknolojidir. Her sanal makine kendi işletim sistemine ve uygulamalarına sahiptir. Ancak, bu sanal makineler doğrudan fiziksel bilgisayarla etkileşim kuramazlar. Bunun yerine, bir hipervizör adı verilen bir yazılım katmanı aracılığıyla fiziksel donanım ile iletişim kurarlar. Hipervizör; işlemciler, bellek ve depolama gibi fiziksel kaynakları sanal makineler arasında paylaştırır ve yönetir. Sanal makineler sayesinde, tek bir fiziksel makine üzerinde farklı işletim sistemleri çalıştırılabilir.

   Docker, containerların yönetildiği açık kaynak kodlu bir yazılımdır. Container ise image kullanılarak oluşturulur. Image, bir uygulamayı çalıştırmak için gerekli olan (kod, kütüphaneler, bağımlılıklar vb.) bilgileri içeren çalıştırılabilir bir yazılım paketidir. Image'lar, container'lar oluşturmak için gereken talimatların bulunduğu bir şablon olarak düşünülebilir.

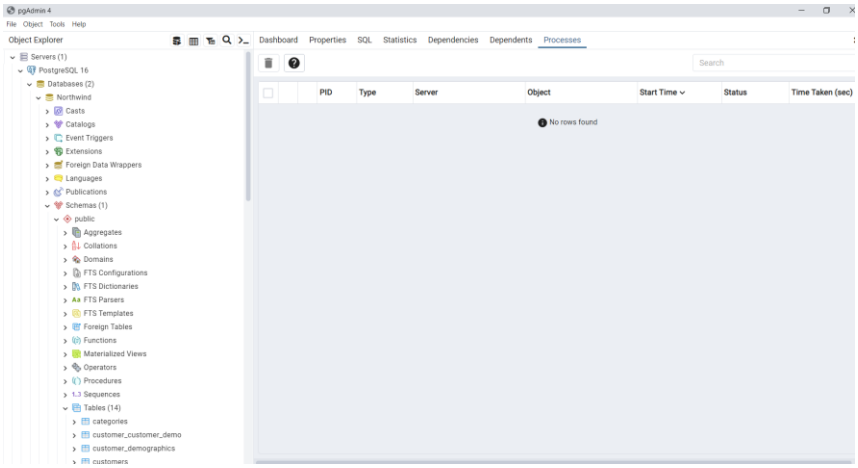
VM, herhangi bir donanımda sanal makine çalıştırmanıza olanak tanır. Docker, herhangi bir işletim sisteminde uygulama çalıştırmanıza olanak tanır. [3][4][5]

4. Docker ile RabbitMQ ve PostgreSQL veya MySQL kurulumu yapın.



Docker Desktop anasayfa

RabbitMQ giriş sayfası



pgAdmin 4 sayfası

5. Docker komutlarını örneklerle açıklayın.

Genel Komutlar

`docker version`: Bu komut ile docker sürümünüzü öğrenebilirsiniz.
`docker info`: Sisteminizde bulunan containerları, ve diğer sistem bilgilerinizi görmenizi sağlar.
`docker system prune`: Sistemde bulunan tüm çalışmayan containerları, kullanılmayan networkleri, kullanılmayan imajları ve build önbelleklerini temizler.
`docker system df`: Docker'ın disk kullanım bilgilerini verir.

Container Komutları

`docker ps`: Çalışan containerları listeler.
`docker ps -a`: Sistem bulunan tüm containerları listeler.
`docker container ls -a`: "docker ps -a" ile aynı işlevi görür.
`docker container ls -aq`: Sistemde bulunan tüm containerların ID'sini listeler.
`docker run --name hello-world hello-world`: hello-world isminde container çalıştırır.
`docker start hello-world`: hello-world isimli containerı başlatır.
`docker stop hello-world`: hello-world isimli containerı durdurur.
`docker logs hello-world`: hello-world isimli containerın loglarını gösterir.
`docker cp "container-id or container-name":path host_path`: İsmi ya da ID'si verilen container'dan host'a dosya kopyalar.
`docker run -d -p 8080:80 "image-name"`: Çalıştırılan containera port numarası verilerek çalıştırabilir. 8080 hostun portunu, 80 ise containerın portunu belirtir. "-d" ile sürekli çalışması sağlanır.
`docker run -dit "image-name" sh`: "-dit" komutu -d -it -tty birleşimidir. Containera interaktiflik ve sözde terminal bağlantısını ekleyerek arka planda çalıştırır.
`docker run -dit --net "network-name" "image-name" sh`: Oluşturulan network ile container oluşturur.
`docker exec -it "container_id" sh`: Uzaktaki containera bir shell bağlantısı ile bağlanmamızı sağlar.
`docker top "container name or id`: Containerda hangi işlemlerin çalıştığını gösterir.
`docker run --rm -it hello-world sh`: Containerı çalıştırır. "--rm" container kapandıktan sonra sil demektir. "-it" "--interactive" ve "--tty" birleşimidir. Containera interaktif bağlantı yapar. "sh" ile de uzak makinedeki containera terminal bağlantısı ekler. "--rm" sadece container interaktif çalışırken olur. "-d" (detach) ile arka planda çalışsın diyemiyoruz.
`docker stats "container name or id"`: Çalışan containerın ne kadar kaynak kullandığını gösterir.
`docker attach "container-name or id"`: Arka planda çalıştırılan containera bağlanmak için kullanılır.
`docker run -d --memory=100m "image-name"`: Oluşturulacak containera 100mb bellek limiti tanımlar.
`docker run -d --cpus="1.5" "image-name`: Sistemde bulunan işlemci çekirdeklerinin 1.5 tanesini kullanmasını sağlar.

`docker run -d --cpuset-cpus="1,4" "image-name"`: Sistemde bulunan işlemcinin 1 ve 4 numaralı işlemcilerini kullanmasını sağlar.
`docker run --env deg1=deneme "image-name"`: deg1 adında ortam değişkeni tanımlar.
`docker container rm "container name or id"`: İsmi ya da ID'i verilen containerı siler.
`docker container rm -f "container name or id"`: İsmi ya da ID'si verilen containerı çalışırken dahi siler.
`docker container prune`: Çalışmayan tüm containerları siler.
`docker ps --filter="exited"`: Sistemde bulunan sadece çalışmayan containerları listeler.
`docker ps --filter="running"`: Sistemde bulunan sadece çalışan containerları listeler.

Image Komutları

`docker image build -t image-name .`: Dockerfile yazılıktan sonra image build işlemini gerçekleştirir.
`docker image build -t "image-name" -f "dockerfile-name" .`: Farklı bir Dockerfile'dan imaj build eder.
`docker images`: Sistemde bulunan imajları listeler.
`docker image rm "image-name"`: İmaj siler.
`docker image prune -a`: Çalışmayan tüm imajları siler.
`docker history "image-name"`: İmaj'ın geçmişini gösterir.

Volume Komutları

`docker volume create firstvolume`: firstvolume adında volume oluşturur.
`docker volume inspect firstvolume`: firstvolume ayrıntılarını verir.
`docker volume ls`: Sistemde bulunan volumeleri listeler.
`docker volume rm "volume-name"`: Adı verilen volumeü siler.
`docker volume prune`: Sistemde bulunan tüm volumeleri siler.

Network Komutları

`docker network create "name"`: Varsayılan network driverı ile network oluşturur.
`docker network ls`: Docker Network objelerini listeler.
`docker network connect "network-name" "container name or id"`: Adı veya ID'si verilen containerı kullanıcı tarafından oluşturulan networke bağlamayı sağlar. Network bağlama işlemi sadece kullanıcı tarafından oluşturulan networkte yapılabilir.
`docker network disconnect "network-name" "container name or id"`: Adı veya ID'si verilen containerı kullanıcı tarafından oluşturulan network ile bağlantısını keser.
`docker network rm "network-name`: Network silmeyi sağlar.

Compose Komutları

`docker-compose ps`: docker-compose ile oluşturulmuş çalışan containerları listeler.

`docker-compose ps -a`: docker-compose ile oluşturulmuş tüm containerları listeler.
`docker-compose up`: Servisleri ayağa kaldırır.Shell'e geri dönmez.
`docker-compose down`: Servisleri durdurur ve siler. yaml dosyasında volume tanımlanmışsa volumleri silmez. "build" ile yaml dosyasından imaj oluşturulmuş ise imaj da silinmez.
`docker-compose up -d`: Servisleri ayağa kaldırır. Shell' geri döner.
`docker-compose config`: yaml dosyasının içeriğini ters sıralama ile listeler.
`docker-compose images`: Oluşturulan servislerin hangi imajlardan oluşturulduğunu listeler.
`docker-compose logs`: Servislerin loglarını görüntüler.
`docker-compose exec "service-name" sh`: Compose ile oluşturulan servise interaktiv shell bağlantısı yapar.
`docker-compose build`: yaml dosyasında build ile imaj oluşturulmuş ise imaj da sonradan yapılan değişiklikleri kayıt eder ve yeni değişikliklerin olduğu imajı kullanılmaya hazır hale getirir.

Swarm Komutları

`docker swarm init`: Komutun çalıştırıldığı makine de swarm aktif edilir.
`docker swarm init --advertise-addr "ip-addr"`: IP adresi verilen makine de swarm aktif edilir.
`docker swarm join-token worker`: Worker eklemek için gerekli olan tokenir getirir.
`docker swarm join-token manager`: Manager eklemek için gerekli olan tokeni getirir.
`docker node ls`: Cluster'da bulun node'ları listeler.
`docker service create --name "service-name" --replicas=5 -p 8080:80 "image-name"`: İsmi service-name olan, 8080 portu servisin 80 portuna yönlendiren ve 5 adet replicası olan servisi oluşturur.
`docker service ps "service-name"`: İsmi verilen servisin task'leri gösterir.
`docker service ls`: Çalışan servisleri listeler.
`docker service inspect "service-name"`: İsmi verilen servisin ayrıntılarını gösterir.
`docker service logs "service-name"`: İsmi verilen servisin loglarını gösterir.
`docker service rm "service-name"`: İsmi verilen servisi siler. Doğrulama yapmadan direkt olarak servisi siler, kullanırken dikkatli olunmalıdır.

Secret Komutları

`docker secret create "secret-name" file`: secret-name adında secret oluşturur.
`docker secret ls`: Sistemde bulunan secretları listeler.
`docker secret inspect "secret-name"`: İsmi verilen secretın ayrıntılarını verir.
`docker secret rm "secret-name"`: İsmi verilen secretı siler.
`echo "deneme" | docker secret create "secret-name"`: Terminal üzerinden secret oluşturur.
`docker service update --secret-rm "secret-name" "service-name"`: İsmi verilen serviste, ismi verilen secretı siler.
`docker service update --secret-add "secret-name" "service-name`: İsmi verilen serviste, ismi verilen secretı ekler.

Logs Komutları

`docker logs "container name or id"`: Container çalıştıktan sonra logları getirir.
`docker logs -f "container name or id"`: Çalışan containerın anlık loglarını gösterir.

Save and Load Komutları

`docker save "image-name" -o "image-name.tar"`: İmajı tar arşivine dönüştürür.
`docker load -i "image-name.tar"`: tar arşivini imaja çevirir.

Docker Commit Komutları

`docker commit "container-id" "image-name"`: Container'daki değişikliklerden yeni bir imaj oluşturur. [6]

6. Microservice ve Monotlith mimarilerini kıyaslayın.

Monolitik mimari, tüm uygulamanın tek, sıkı bir şekilde entegre edilmiş bir birim olarak oluşturulduğu, genellikle "monolit" olarak adlandırılan geleneksel bir yazılım tasarım modelidir. Monolitik bir uygulamada tüm bileşenler ve modüller aynı kod tabanını, veritabanını ve çalışma zamanı ortamını paylaşır.

Mikroservis mimarisi, bir uygulamayı daha küçük, bağımsız olarak dağıtılabilen hizmetlerden oluşan bir koleksiyona ayırır. Her hizmet belirli işlevlerden sorumludur ve birbirleriyle iletişim kurabilirler.

Monolitik Mimarinin Güçlü Yönleri:

Hata Ayıklama ve Test Etme Kolaylığı: Mikroservis mimarisinden farklı olarak, monolitik uygulamaların hata ayıklanması ve test edilmesi daha basittir. Çünkü monolitik bir uygulama tek bir bütün halindedir, bu nedenle uçtan uca testler daha hızlı bir şekilde gerçekleştirilebilir.

Kolay Dağıtım: Monolitik uygulamaların tek parça olması, onları dağıtım açısından kolay hale getirir. Tek bir parçayı dağıtmak, onlarca hizmeti dağıtmaktan daha basittir.

Geliştirmesi Kolaydır: Monolitik bir uygulamayı modüler bir şekilde yazmak, her bir modülü ayrı bir hizmet olarak yazmaktan daha basittir. Bu, genel prensiplere uyarak daha basit geliştirmeyi sağlar.

Monolitik Mimarinin Zayıf Yönleri:

Karmaşıklık: Bir monolitik uygulama büyüdükçe karmaşık hale gelir ve yönetilmesi zorlaşır. Ayrıca, karmaşık bir kod yapısını yönetmek zordur ve kodun büyümesi karmaşıklığı artırır.

Değişiklik Yapma Zorluğu: Büyük ve karmaşık bir uygulamada değişiklik yapmak zordur. Herhangi bir kod değişikliği tüm sistemi etkiler, bu nedenle bütün parçaların kontrol edilmesi gerekir ve bu da geliştirme sürecini uzatır.

Ölçeklenebilirlik Sorunu: Monolitik uygulamalarda bileşenleri bağımsız olarak ölçeklendirmek mümkün değildir. Tüm uygulama birlikte ölçeklenmelidir.

Yeni Teknoloji Entegrasyonu Güçlüğü: Monolitik bir uygulamaya yeni bir teknoloji entegre etmek zordur ve tüm uygulamanın yeniden geliştirilmesini gerektirebilir. Bu da yeniliklere adapte olmayı zorlaştırabilir ve kayıplara yol açabilir.

Microservice Mimarisinin Güçlü Yönleri:

Bağımsız Bileşenler: Her servis, bağımsız olarak dağıtılabilir ve güncellenebilir, bu da esneklik sağlar. Bir servisteki hata, sadece o servisi etkiler ve tüm uygulamayı etkilemez. Yeni özellikler eklemek, monolitik yapıdan daha kolaydır.

Anlaşılabilirlik: Microservice mimarisine göre tasarlanmış bir uygulamanın daha küçük ve basit bileşenlere ayrılması, yönetilmesi ve anlaşılması daha kolaydır. Her servis, sadece belirli bir sorumlulukla ilgilidir.

Daha İyi Ölçeklenebilirlik: Her servis, bağımsız olarak ölçeklendirilebilir, bu da tüm uygulamanın ölçeklenmesini gerektirmez. Bu, zaman ve maliyet açısından önemli bir avantajdır.

Teknoloji Çeşitliliği: Ekipler, servislerin geliştirileceği teknolojileri tamamen seçebilir ve istedikleri teknolojiyi kullanabilirler. Her servis için farklı teknolojiler veya veritabanları kullanılabilir.

Daha Yüksek Çeviklik Seviyesi: Bir servisteki hata, tüm uygulamayı değil sadece ilgili servisi etkiler. Bu nedenle değişiklikler ve testler daha düşük risklerle gerçekleştirilir.

Yeniden Kullanılabilirlik: Bağımsız olarak geliştirilen bir servis, başka bir projede kullanılabilir ve ufak değişikliklerle adapte edilebilir. Bu, geliştirme zamanından ve maliyetinden tasarruf sağlar.

Microservice Mimarisinin Zayıf Yönleri:

Karmaşıklık: Microservice mimarisi, dağıtık bir sistem olduğu için tüm modüller ve veritabanları için ayrı yapılandırma gerektirir.

Sistem Dağıtımı: Birden fazla servis ve veritabanından oluşan karmaşık bir sistem olduğundan, tüm bağlantıların dikkatle ele alınması gerekir. Her servisin ayrı bir dağıtım süreci vardır.

Yönetim ve İzlenebilirlik Zorluğu: Birden fazla durumla ilgilenmek gerekebilir ve harici yapılandırma, metrik izleme ve sağlık kontrol mekanizmaları gibi ekstra özellikler gerekebilir.

Test Etmenin Zorluğu: Bağımsız olarak konuşlandırılan çok sayıda servis, test sürecini zorlaştırır ve yönetimi karmaşık hale getirebilir. [7]

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

API Gateway

API Gateway, bir mikroservis mimarisi veya bulut tabanlı uygulama ortamında API (Application Programming Interface) trafiğini yönetmek ve kontrol etmek için kullanılan bir yazılım bileşenidir. API Gateway, gelen istekleri alır, doğrular, yönlendirir ve gerektiğinde dönüştürür, ardından ilgili hizmetlere iletir ve gelen cevapları geri döndürür. API Gateway, bir uygulamanın tüm dış API isteklerini yönetmek için tek bir giriş noktası sağlar. Bu, uygulamanın içindeki hizmetlerin veya mikroservislerin doğrudan dış dünyayla iletişim kurmasını önler. Aşağıdaki durumlar için kullanılabilir:

Kimlik Doğrulama ve Yetkilendirme: API Gateway, gelen istekleri doğrulama ve yetkilendirme işlemlerinden geçirebilir. Kullanıcı kimlik doğrulaması yapabilir, yetkilendirme kontrolleri gerçekleştirebilir ve erişim izinlerini yönetebilir.

Trafik Yönlendirme ve Yönetimi: API Gateway, gelen istekleri belirli hizmetlere yönlendirir ve gerektiğinde farklı hizmetler arasında trafik yönetimi yapabilir. Bu, yük dengeleme, yönlendirme politikaları ve hizmetler arası trafik kontrolü gibi işlevleri içerebilir.

Logging: Servislere yapılan istekler hakkında detaylı loglamalar gerçekleştirilebilir ve böylece hangi servis, kim tarafından, ne kadar yoğunlukta işlevsellik gösteriyor vs. gibi istatistiksel bilgiler edinilebilir.

Güvenlik Özellikleri: API Gateway, gelen istekleri güvenlik tehditlerine karşı denetleyebilir ve güvenlik politikalarını uygulayabilir. Bu, kötü niyetli saldırılara karşı koruma sağlar ve uygulamanın güvenliğini artırır. [8]

Service Discovery

Service Discovery, dağıtık bir sistemdeki hizmetlerin dinamik olarak bulunması ve iletişim kurulması için kullanılan bir yöntemdir. Genellikle bulut tabanlı veya mikroservis mimarisi gibi ortamlarda kullanılır.

Service Discovery, bir ağdaki hizmetlerin otomatik olarak keşfedilmesini ve bunlarla iletişim kurulmasını sağlar. Bu, bir hizmetin IP adresi, bağlantı noktası, sağlığı veya diğer meta veriler gibi bilgilerin dinamik olarak bulunmasını içerir.

"Service discovery" için iki temel desen bulunmaktadır: "Client-Side Service Discovery" ve "Server-Side Service Discovery".

Client-Side Service Discovery:

Bu yapıda, istemci (client) hizmet kayıt defteri (service registry) ile iletişime geçer ve belirli bir alan adının adresini talep eder. Hizmet kayıt defterinde kaydedilmiş ve erişilebilir durumda olan mikroservis örneklerinin IP adreslerini alır. Daha sonra istemci, bu mikroservisler arasından en uygun olanı seçmek için bir yük dengeleme algoritması kullanır ve mikroservis ile iletişim kurar.

Server-Side Service Discovery:

Server-Side Service Discovery'de, istemci (client) bir yük dengeleyici (load balancer) ile iletişime geçer ve isteğini buraya iletir. Yük dengeleyici, hizmet kayıt defteri (service registry) ile iletişim kurarak erişilebilir durumdaki örneklerin listesini alır. Daha sonra yük dengeleyici, kendi algoritmasına göre en uygun gördüğü örneğe isteği yönlendirir. Bu şekilde, istemci doğrudan hizmet kayıt defterine sormaz, bunun yerine yük dengeleyici aracılığıyla hizmete erişir. [9]

Load Balancer

Load balancer, gelen trafiği birden çok sunucuya veya kaynağa dağıtan bir hizmettir. Çoğu durumda, iki veya daha fazla web sunucusunun önünde bulunur ve ağ trafiğini bunlar arasında dağıtır. Bu, sunucu kaynaklarının en iyi şekilde kullanılmasına ve kullanıcının cihazına verimli içerik dağıtımı yapılmasına yardımcı olur.

Temel olarak, bir load balancer API'ı, isteklerin doğru zamanda doğru sunucuya gönderilmesini sağlar; istekleri eşit şekilde dağıtarak herhangi bir sunucu üzerindeki yükü etkili bir şekilde azaltır.

Load balancerın birincil rolü, iki veya daha fazla sunucu arasındaki yükü dengelemektir. Bunu, arka uç sunucularının sağlığını izleyerek ve trafiği buna göre dağıtarak yapar. Ayrıca yük dengeleyici aşağıdakiler gibi başka önemli işlevleri de sağlayabilir:

SSL Offloading – HTTPS trafiğinin şifrelenmesi/şifre çözülmesinin yönetilmesi sorumluluğunu üstlenir.

HTTP Compression – Ağ üzerinden gönderilen veri miktarını azaltmak için web sayfalarını sıkıştırır.

Content Caching – Sık kullanılan içeriği bir önbellekte saklar, böylece gerektiğinde hızla alınabilir. [8]

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

   Hibernate

   Hibernate, veritabanı işlemlerinde kullanılan bir framework'tür. ORM (Object Relational Mapping) aracı olarak kullanılır. ORM, uygulamamızdaki nesnelerle ilişkisel veritabanları arasında bağ kurmamızı sağlayan bir yapıdır. Yerel SQL sorguları yazmadan verileri saklamayı, almayı ve değiştirmeyi kolaylaştırır. [10]

   Örnek;

```
// Importing required classes
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "song")

// POJO class
public class Song {

    @Id @Column(name = "songId") private int id;

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

    @Column(name = "singer") private String artist;

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }

    public String getSongName() { return songName; }

    public void setSongName(String songName)
    {
        this.songName = songName;
    }

    public String getArtist() { return artist; }

    public void setArtist(String artist)
    {
```

```
        this.artist = artist;
    }
}

// Java Program to Illustrate App File

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

// Main class
public class App {

    // Main driver method
    public static void main(String[] args)
    {

        // Create Configuration
        Configuration configuration = new Configuration();
        configuration.configure("hibernate.cfg.xml");
        configuration.addAnnotatedClass(Song.class);

        // Create Session Factory and auto-close with try-with-
resources.
        try (SessionFactory sessionFactory
                = configuration.buildSessionFactory()) {

            // Initialize Session Object
            Session session = sessionFactory.openSession();

            Song song1 = new Song();

            song1.setId(1);
            song1.setSongName("Broken Angel");
            song1.setArtist("Akon");

            session.beginTransaction();

            // Here we have used
            // persist() method of JPA
            session.persist(song1);

            session.getTransaction().commit();
        }
    }
}
     //[11]
```

JPA, Java Persistence API'nin kısaltmasıdır. JPA, Java platformunda ORM (Object Relational Mapping) için standart bir API'dir. Bu API, Java uygulamalarının ilişkisel veritabanlarıyla etkileşimini kolaylaştırır ve veri tabanı işlemlerini nesne odaklı bir şekilde yapmayı sağlar.

JPA, Java uygulamalarında veritabanı işlemlerini gerçekleştirmek için kullanılan bir araçtır. İlişkisel veritabanlarında saklanan verileri Java nesnelerine dönüştürmeyi ve Java nesnelerini veritabanına kaydetmeyi sağlar. Bu sayede, geliştiriciler veritabanı işlemlerini SQL sorgularıyla değil, Java koduyla yapabilirler. JPA, kendi başına bir şey yapamaz, bir implementation'a yani uygulamak için bir araca ihtiyaç duyar.

Bazı JPA implementasyonları:

- Hibernate JPA
- EclipsLink
- Open JPA [12]

Spring Data

Spring Data JPA, JPA tabanlı (Java Persistence API) depolarının kolayca uygulanmasını kolaylaştırır. Veri erişim teknolojilerini kullanan Spring destekli uygulamalar oluşturmayı kolaylaştırır.

Spring Data, veritabanı ve diğer veri depolama teknolojileri ile etkileşimi kolaylaştırır ve geliştiricilere daha az kod yazarak daha fazla iş yapma imkanı sunar. Spring Data'nın temel amacı, veri erişimini kolaylaştırmak ve geliştirme sürecini hızlandırmaktır.

@Query anatosyonunu kullanarak kompleks sorgular yazmamıza olanak tanır.

```
import javax.persistence.*;

@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private double price;

    // Getters and setters
}
```

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import java.util.List;

public interface ProductRepository extends
JpaRepository<Product, Long> {

    // Example of a custom query using @Query annotation
    @Query("SELECT p FROM Product p WHERE p.price > ?1")
    List<Product> findByPriceGreaterThan(double price);
}

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    public List<Product> getProductsByPriceGreaterThan(double
price) {
        return productRepository.findByPriceGreaterThan(price);
    }
}
//[13]
```

KAYNAKÇA

[1] https://gokhana.medium.com/microservice-mimarisinde-servisler-aras%C4%B1-i%CC%87leti%C5%9Fim-ve-event-driven-mimari-e02cdf87e5fe#:~:text=Senkron%20%C4%B0leti%C5%9Fim,kuran%20bir%C3%A7ok%20farkl%C4%B1%20servis%20vard%C4%B1r.

[2] https://aws.amazon.com/tr/compare/the-difference-between-rabbitmq-and-kafka/

[3] https://medium.com/software-development-turkey/dockera-giri%C5%9F-virtual-machines-vs-containers-ea67848dfefb

[4] https://www.ibm.com/topics/virtual-machines?utm_medium=OSocial&utm_source=Youtube&utm_content=SOFWW&utm_id=YT-101-Docker-vs-VMs

[5] https://aws.amazon.com/tr/compare/the-difference-between-docker-vm/#:~:text=VM%2C%20herhangi%20bir%20donan%C4%B1mda%20sanal,sisteminde

%20uygulama%20%C3%A7al%C4%B1%C5%9Ft%C4%B1rman%C4%B1za%20olanak%20tan%C4%B1r.

[6] https://github.com/akiffeyzioglu/docker-notes?tab=readme-ov-file#Docker-Komutlar%C4%B1

[7] https://gokhana.medium.com/monolitik-mimari-ve-microservice-mimarisi-aras%C4%B1ndaki-farklar-bd89ac5b094a

[8] https://blog.hubspot.com/website/api-gateway-vs-load-balancer

[9] https://medium.com/i%CC%87yi-programlama/microservice-mimarilerinde-service-discovery-7a6ebceb1b2a

[10] https://medium.com/thefreshwrites/exploring-the-most-common-java-libraries-frameworks-6c7ae547d23c

[11] https://www.geeksforgeeks.org/hibernate-example-using-jpa-and-mysql/

[12] https://emrllhyildirim.medium.com/jpa-ve-hibernate-aras%C4%B1ndaki-fark-nedir-faa677fa467

[13] https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa