Q1 -

Senkron iletişim ve asenkron iletişim, bilgi veya veri aktarımının gerçekleşme şekillerini ifade eder.

1. Senkron İletişim: Senkron iletişimde, veri gönderme ve alma işlemleri belirli bir zaman çerçevesinde gerçekleşir ve iletişim kanalındaki tarafların eş zamanlı olarak işbirliği yapması gerekmektedir. Gönderici veri gönderdiğinde, alıcı veriyi alırken belirli bir süre bekleme sürecine girer. Gönderici, verinin alıcının tarafında güvenli bir şekilde işlendiğini ve alındığını doğruladıktan sonra yeni bir işlem gerçekleştirebilir. Örneğin, telefon görüşmeleri senkron bir iletişim örneğidir. Bir kişi konuşurken diğer kişi bekler ve cevap verir.

2. Asenkron İletişim: Asenkron iletişimde, veri gönderme ve alma işlemleri belirli bir zaman çerçevesine bağlı değildir ve iletişim kanalındaki tarafların eş zamanlı olarak işbirliği yapması gerekmez. Gönderici veriyi gönderir ve işlemi tamamlar, ardından alıcı veriyi aldığında veya işlediğinde geri bildirimde bulunabilir. Gönderici, verinin alıcıya ulaşıp ulaşmadığını veya alıcı tarafından işlenip işlenmediğini beklemeden başka işlemler yapabilir. E-posta iletişimi asenkron bir iletişim örneğidir. Bir kişi e-posta gönderir ve gönderdikten sonra alıcının e-postayı okumasını veya yanıtlamasını bekler, ancak gönderici bu süreçte beklemek zorunda değildir.

Özetle, senkron iletişimde işlemler belirli bir sıraya göre ve eş zamanlı olarak gerçekleşirken, asenkron iletişimde işlemler zaman bağımsızdır ve işbirliği gerektirmez.

Q2 –

İkisi de mesaj sıralama sistemleridir.

	RabbitMQ	**Kafka**
Kullanım Senaryoları	RabbitMQ, genellikle geleneksel mesaj sıralama kullanım senaryoları için tercih edilir. Bu, RPC (Remote Procedure Call), iş kuyrukları ve olay tabanlı sistemler gibi durumları içerir. RabbitMQ, işletmelerin yaygın olarak kullandığı işlenen gönderme-almada kesinlik gerektiren durumlar için uygundur.	Kafka, genellikle büyük ölçekli, yüksek hacimli veri akışları ve gerçek zamanlı işleme senaryoları için tercih edilir. Büyük veri akışlarını işlemek, gerçek zamanlı veri analitiği ve loglama gibi durumlar için Kafka daha uygundur.
Mimari	RabbitMQ, yayılma tabanlı (publish-subscribe) ve kuyruk tabanlı (queue-based) mesajlaşma için esnek bir mimari sunar. Bu, farklı senaryolar için farklı iletişim modellerinin kullanılmasına izin verir.	Kafka, anahtar-kıymet çiftleri (key-value pairs) şeklinde sıralanmış kayıtların (log records) dağıtılmış bir akışıdır. Kafka, ana hat odaklı mimariyi benimser ve temel olarak veri akışı işleme üzerine odaklanır.
Dayanıklılık ve İş Garantisi	RabbitMQ, mesajların güvenilir bir şekilde teslim edilmesini sağlamak için kuyruk mekanizmaları kullanır. Mesajlar bir kuyrukta bekler ve alıcı mesajları aldığında kuyruktan silinir.	Kafka, disk üzerinde kalıcı verileri saklar ve mesajların daha kalıcı bir şekilde saklanmasını sağlar. Ayrıca, mesajların yayımlanması ve tüketilmesi arasında tutarlılık sağlar.

Ölçeklenebilirlik	RabbitMQ, ölçeklenebilirlik konusunda Kafka'ya göre daha sınırlıdır. RabbitMQ, daha geleneksel ve orta ölçekli uygulamalarda kullanılmaya daha uygun bir şekilde tasarlanmıştır.	Kafka, yüksek ölçeklenebilirlik sağlar. Kafka, birden fazla broker üzerinde dağıtılmış bir şekilde çalışabilir ve veri akışlarını kolayca ölçeklendirebilir.
İşleme Süreleri	RabbitMQ, genellikle düşük gecikme sürelerine odaklanır ve mesajların hızlı bir şekilde işlenmesini sağlar	Kafka, genellikle yüksek performans ve düşük gecikme süreleri sağlar, ancak düşük gecikme sürelerinden daha fazla verimlilik ve ölçeklenebilirlik avantajı sunar.

Ayrıntılı Bilgilendirme : https://aws.amazon.com/tr/compare/the-difference-between-rabbitmq-and-kafka/

Q3 –

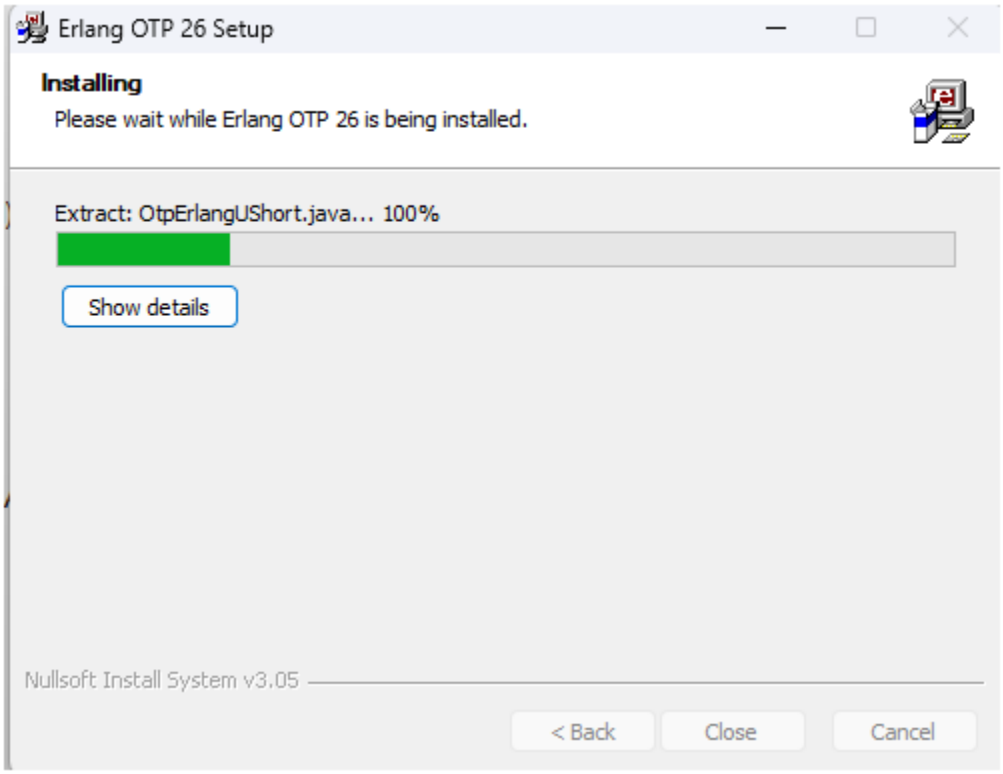
Docker ve sanal makine (Virtual Machine - VM), yazılım uygulamalarını çevrelerinden bağımsız bir şekilde çalıştırmak için kullanılan iki farklı teknolojidir. Her ikisi de farklı yaklaşımlarla çalışır ve farklı kullanım senaryolarına sahiptir.

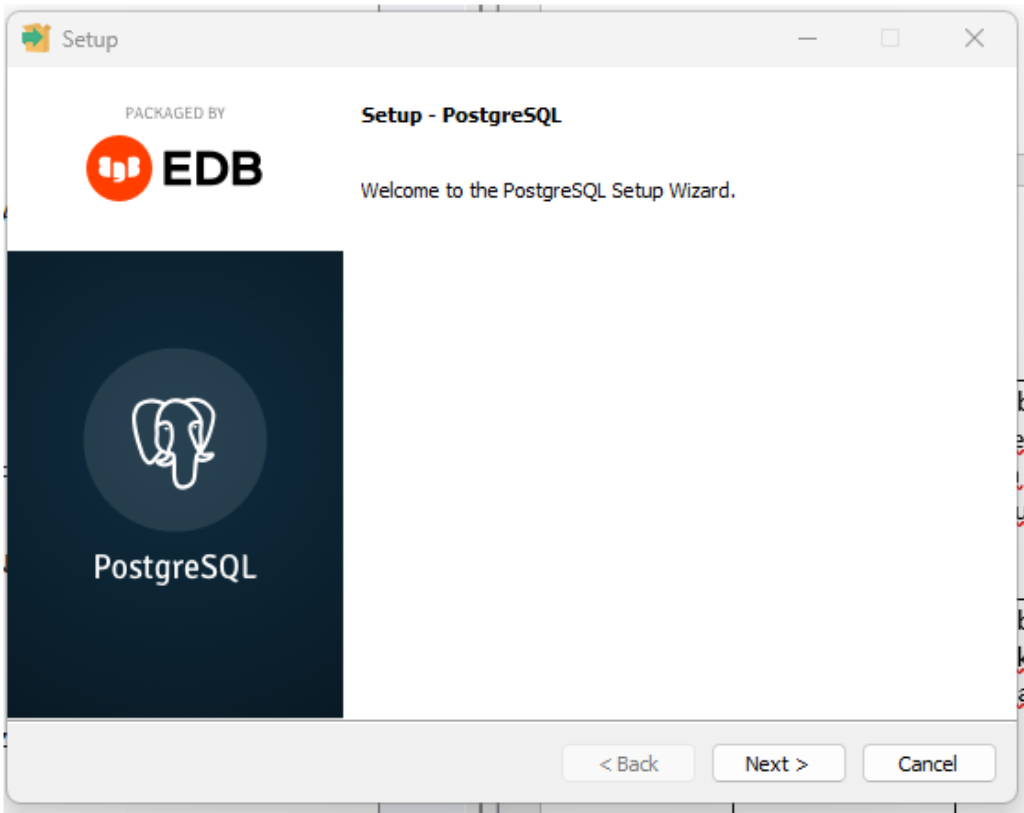
1. **Docker:**
    - Docker, yazılım uygulamalarını konteynır adı verilen hafif, taşınabilir ve kapsayıcı bir birimde çalıştırmak için kullanılan bir platformdur.
    - Docker, uygulamaların gereksinim duyduğu tüm bağımlılıkları ve ortamı bir konteyner içerisinde paketler. Bu konteynerler, işletim sistemi çekirdeğini paylaşır, ancak uygulamaların kendine özgü kütüphaneleri ve diğer bağımlılıkları içerir.
    - Docker konteynerleri, hızlı başlatma süreleri ve düşük sistem kaynakları kullanımıyla bilinir. Aynı donanım üzerinde birden çok Docker konteyneri çalıştırılabilir.
2. **Sanal Makine (Virtual Machine - VM):**
    - Sanal makine, fiziksel bir bilgisayar üzerinde, bir veya birden fazla sanal işletim sistemi çalıştırmak için kullanılan yazılım tabanlı bir emülatördür.
    - Sanal makineler, fiziksel donanım üzerinde bir işletim sistemi gibi davranır ve her biri kendi işletim sistemi, uygulamaları ve bağımsız ayarları olan sanal bir ortam sağlar.
    - Her sanal makine, genellikle tam bir işletim sistemi yükler, bu da daha fazla bellek ve işlemci kullanımına neden olur. Birden fazla sanal makine çalıştırmak, donanım kaynaklarının daha fazla paylaşılmasını gerektirir ve bazen performans kayıplarına neden olabilir.

Docker ve sanal makine arasındaki temel fark, Docker'ın daha hafif ve taşınabilir konteynerler kullanarak uygulamaları izole etmesi ve çalıştırmasıdır, sanal makineler ise daha fazla kaynak kullanımına ve

genellikle daha yüksek başlatma sürelerine sahiptir. Her teknolojinin farklı avantajları ve kullanım senaryoları vardır ve hangisinin tercih edileceği, projenin ihtiyaçlarına ve gereksinimlerine bağlıdır.

Q4-

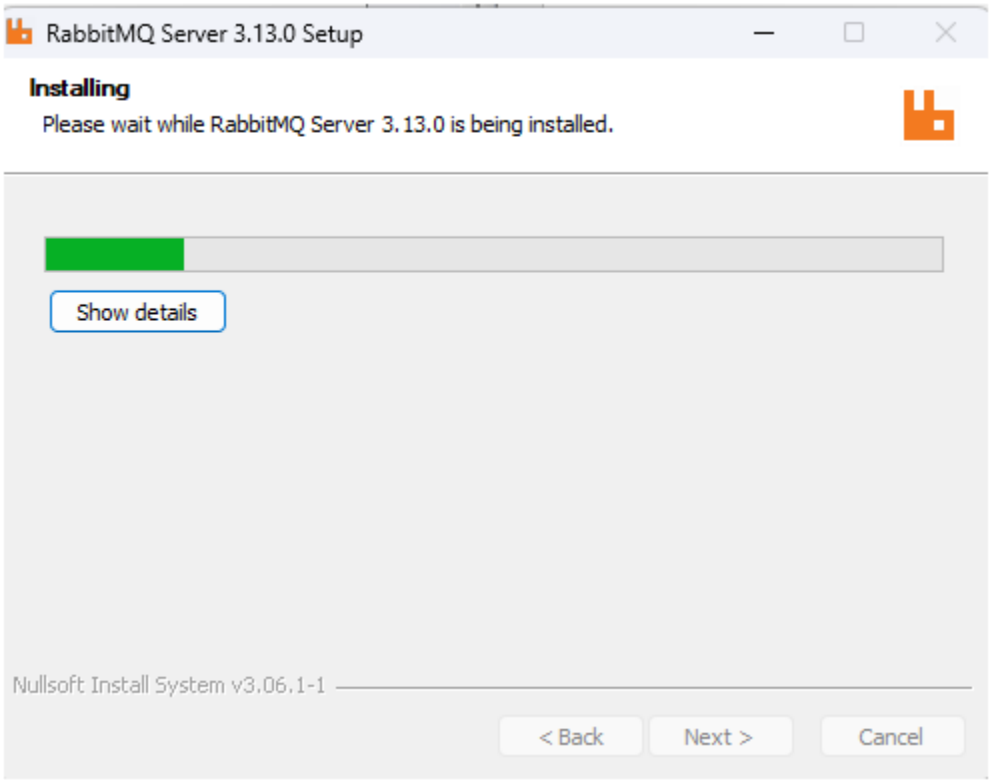


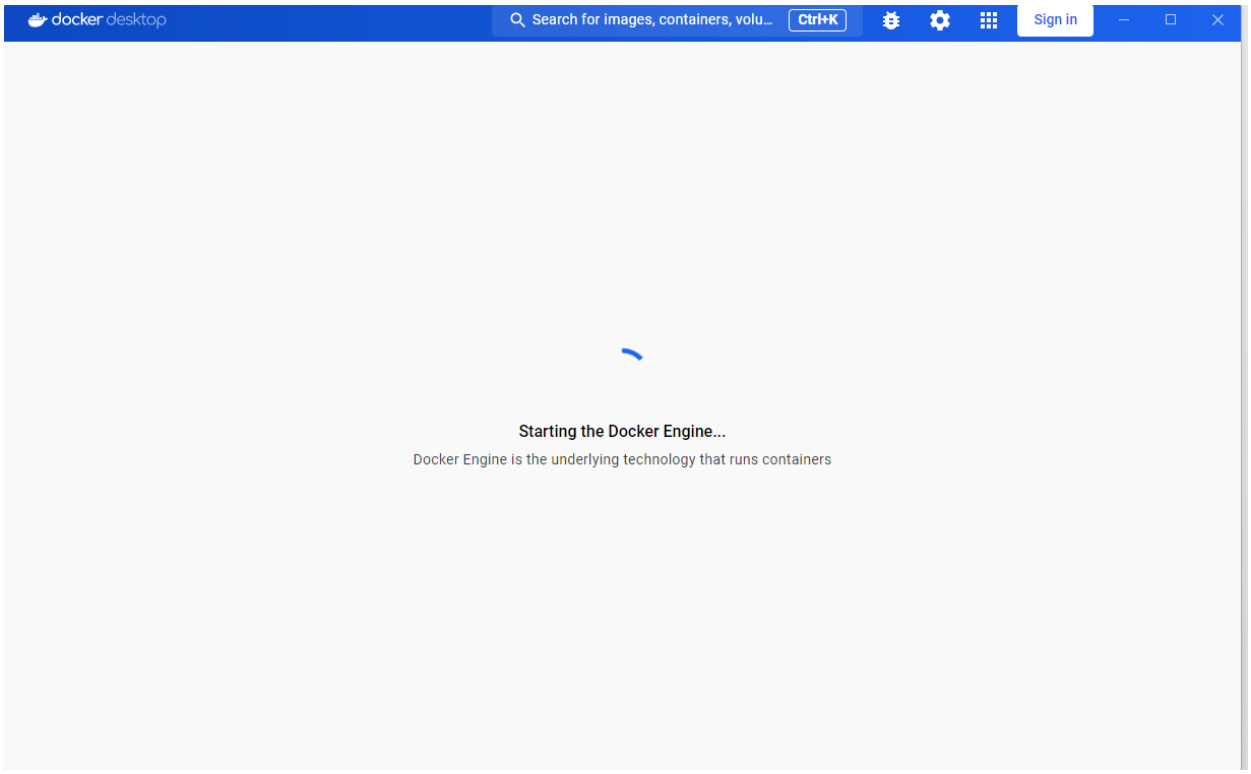


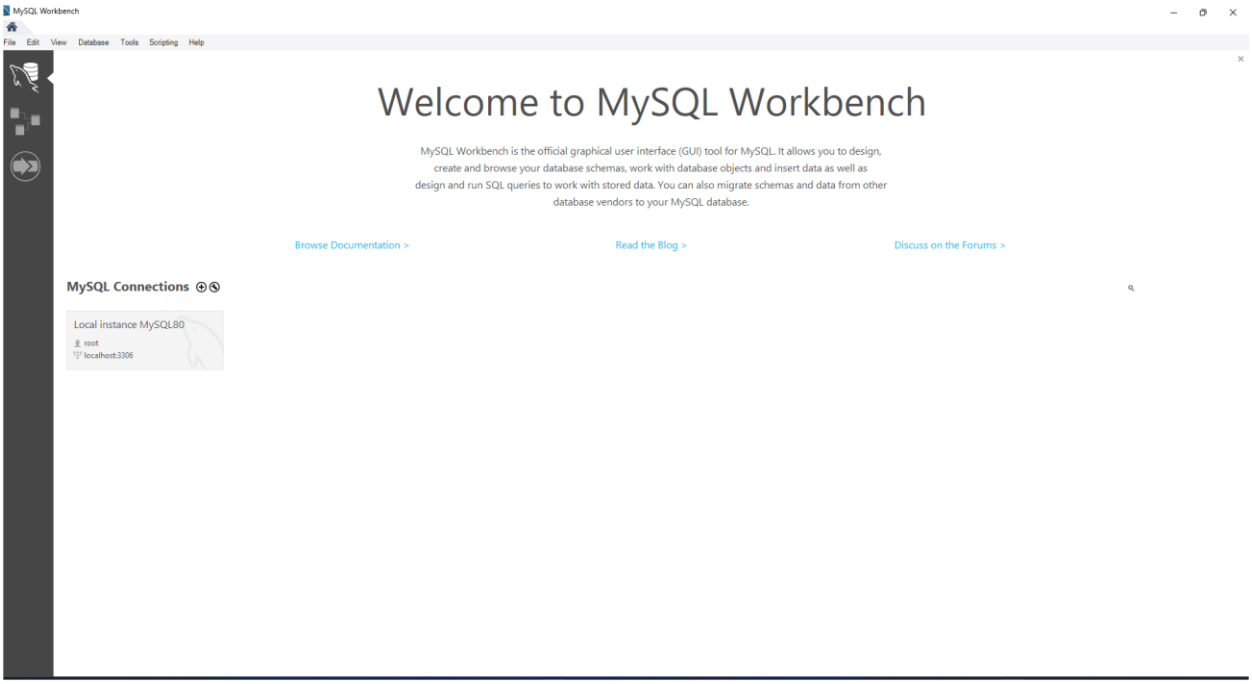
bitMQ, ölçeklenebi
daha sınırlıdır. Rab
ölçekli uygulamala
un bir şekilde tasar

bitMQ, genellikle d
klanır ve mesajların
ar

point'i yazın.




docker desktop
Search for images, containers, volu...
Ctrl+K
Sign in
Starting the Docker Engine...
Docker Engine is the underlying technology that runs containers


MySQL Workbench
File Edit View Database Tools Scripting Help
Welcome to MySQL Workbench
MySQL Workbench is the official graphical user interface (GUI) tool for MySQL. It allows you to design, create and browse your database schemas, work with database objects and insert data as well as design and run SQL queries to work with stored data. You can also migrate schemas and data from other database vendors to your MySQL database.
Browse Documentation >
Read the Blog >
Discuss on the Forums >
MySQL Connections
Local instance MySQL80
root
localhost:3306

Q5 –

Örnek docker komutları aşağıda verilmiştir. Bu cheetsheet kullanılarak Windows cmd ile container oluşturulabilir, image oluşturulabilir, dockerhub'a bunlar pushlanabilir.

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

INSTALLATION

Docker Desktop is available for Mac, Linux and Windows
https://docs.docker.com/desktop

View example projects that use Docker
https://github.com/docker/awesome-compose

Check out our docs for information on using Docker
https://docs.docker.com

IMAGES

Docker images are a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Build an Image from a Dockerfile
```
docker build -t <image_name>
```

Build an Image from a Dockerfile without the cache
```
docker build -t <image_name> . --no-cache
```

List local images
```
docker images
```

Delete an Image
```
docker rmi <image_name>
```

Remove all unused images
```
docker image prune
```

DOCKER HUB

Docker Hub is a service provided by Docker for finding and sharing container images with your team. Learn more and find images at https://hub.docker.com

Login into Docker
```
docker login -u <username>
```

Publish an image to Docker Hub
```
docker push <username>/<image_name>
```

Search Hub for an image
```
docker search <image_name>
```

Pull an image from a Docker Hub
```
docker pull <image_name>
```

GENERAL COMMANDS

Start the docker daemon
```
docker -d
```

Get help with Docker. Can also use –help on all subcommands
```
docker --help
```

Display system-wide information
```
docker info
```

CONTAINERS

A container is a runtime instance of a docker image. A container will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Create and run a container from an image, with a custom name:
```
docker run --name <container_name> <image_name>
```

Run a container with and publish a container's port(s) to the host.
```
docker run -p <host_port>:<container_port> <image_name>
```

Run a container in the background
```
docker run -d <image_name>
```

Start or stop an existing container:
```
docker start|stop <container_name> (or <container-id>)
```

Remove a stopped container:
```
docker rm <container_name>
```

Open a shell inside a running container:
```
docker exec -it <container_name> sh
```

Fetch and follow the logs of a container:
```
docker logs -f <container_name>
```

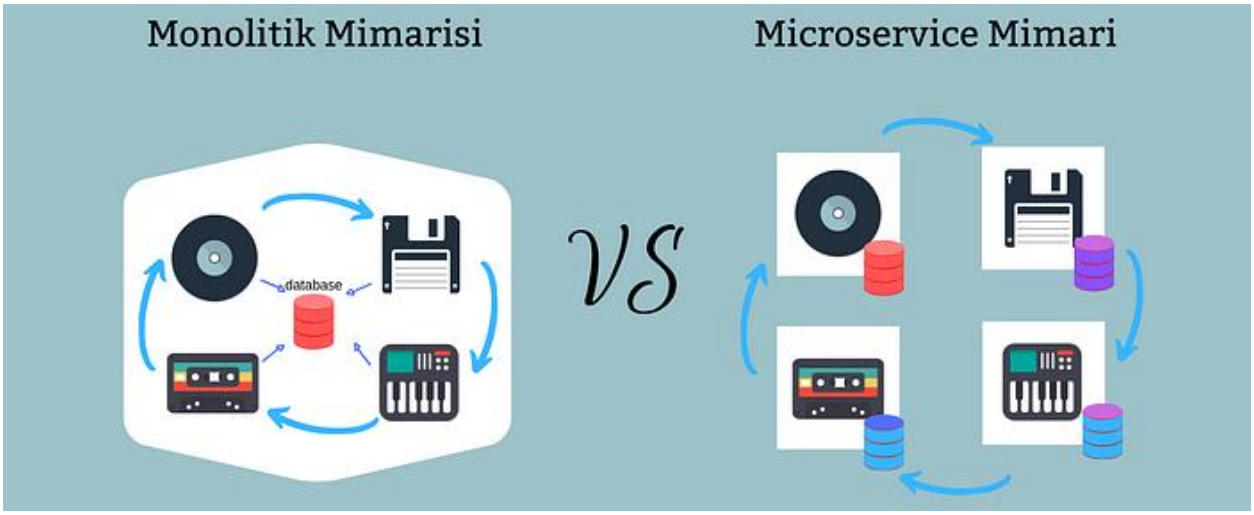
To inspect a running container:
```
docker inspect <container_name> (or <container_id>)
```

To list currently running containers:
```
docker ps
```

List all docker containers (running and stopped):
```
docker ps --all
```

View resource usage stats
```
docker container stats
```

Q6 –



Şekil 1. Monolıtık vs Mıcroservıce Mımarı Referans

1. **Monolithic (Monolitik) Mimariler**:
    - Monolitik mimaride, tüm uygulama tek bir büyük ve tek bir kod tabanında geliştirilir, derlenir ve dağıtılır.
    - Bir monolitik uygulama, genellikle tek bir veritabanı ve tek bir kullanıcı arayüzü ile birlikte gelir.
    - Kod tabanı büyüdükçe, uygulamanın bakımı zorlaşabilir ve kod karmaşık hale gelebilir. Bir hata bulunduğunda, tüm uygulamayı etkileyebilir.
    - Monolitik mimaride, bir bölümdeki değişikliklerin diğer bölümleri etkileme riski vardır.
2. **Microservice Mimariler**:
    - Microservice mimarisinde, bir uygulama, küçük, bağımsız ve özgül işlevselliği olan birçok küçük hizmete bölünür.
    - Her mikroservis, kendi veritabanına sahip olabilir ve kendi API'sini sunabilir.
    - Her mikroservis, bağımsız olarak geliştirilir, dağıtılır ve ölçeklendirilir. Bu, daha hızlı geliştirme döngüleri ve daha hızlı dağıtımlar sağlar.
    - Mikroservisler, hataları izole etme, uygulamayı daha iyi ölçekleme ve farklı teknolojileri kullanma esnekliği gibi avantajlar sağlar.
    - Ancak, mikroservis mimarisi, birden çok hizmetin koordinasyonunu ve dağıtımını yönetmek için karmaşıklık ekler.

Monolitik uygulamalar, genellikle tek bir kod tabanında geliştirilen ve dağıtılan büyük uygulamalardır. Monolitik mimaride, tüm kod tabanı tek bir yapı içinde birleştirilir ve tüm işlevler bir arada bulunur. Bu

nedenle, bir değişiklik yapmak veya bir hata düzeltmek için genellikle tüm uygulamayı güncellemek gerekir. Monolitik mimariler, başlangıçta basitlik ve tek bir yerde yönetim kolaylığı sağlasa da, büyüdükçe bakımı ve ölçeklendirmesi zorlaşabilir.

Mikroservisler mimarisinde ise uygulama, küçük ve bağımsız hizmetlere bölünür. Her bir hizmet, belirli bir işlevselliği sağlar ve kendi veri tabanına veya hatta teknolojisine sahip olabilir. Bu, her hizmetin bağımsız olarak geliştirilmesini, dağıtılmasını ve ölçeklendirilmesini sağlar. Mikro servis mimarisi, esneklik, ölçeklenebilirlik ve hızlı geliştirme döngüleri sağlar. Ancak, bir mikro servis mimarisinin yönetimi daha karmaşık olabilir ve servisler arasındaki iletişimi ve koordinasyonu sağlamak gerekebilir.

Özetle, monolitik mimari daha basit ve daha az karmaşıktır, ancak büyüdükçe bakımı ve ölçeklendirilmesi zorlaşabilir. Mikro servis mimarisi ise daha esnek ve ölçeklenebilir olmasına rağmen, yönetilmesi daha karmaşıktır. Hangi mimarinin kullanılacağı, projenin gereksinimlerine, büyüklüğüne ve ekibin yeteneklerine bağlıdır. Referans

Q7-

1. **API Gateway**:
    - API Gateway, gelen istekleri alır ve uygulamanın arka kısmındaki mikroservislere yönlendirir. Bir API Gateway, birçok farklı istemci (örneğin web tarayıcıları, mobil uygulamalar) iletişim kurduğunda tek bir noktadan erişilebilir ve yönetilebilir bir API sunar. API Gateway, gelen istekleri doğrulama, yetkilendirme, güvenlik kontrolleri, trafik yönlendirme, protokol dönüşümü ve hata yönetimi gibi işlevleri yerine getirebilir. Örnek olarak, gelen istekleri uygun mikroservislere yönlendiren ve çıktıları birleştiren bir API Gateway, birden çok mikroservis arasında karmaşık bir iletişimi basitleştirebilir.
2. **Service Discovery**:
    - Service Discovery, bir mikroservis mimarisinde, uygulamadaki farklı hizmetlerin yerlerini bulmaya ve iletişim kurmaya yardımcı olan bir sistemdir. Mikroservis mimarisinde, hizmetler sürekli olarak ölçeklendirilebilir ve dağıtılabilir. Bu nedenle, bir hizmetin yerini (IP adresi ve port numarası gibi) sabit olarak bilmek yerine, dinamik olarak bulmak önemlidir. Service Discovery, hizmetlerin kaydını ve güncel konumlarını saklar. İstemciler, hizmetlerin yerini sorgulayabilir ve iletişim kurabilir. Örneğin, bir mikroservis, kendini Service Discovery sistemine kaydedebilir ve bir istemci istek yaptığında Service Discovery üzerinden hizmetin yerini bulabilir.
3. **Load Balancer**:
    - Load Balancer, gelen istekleri birden fazla sunucu veya hizmet arasında eşit bir şekilde dağıtmak için kullanılan bir sistemdir.
    - Load Balancer, yük dengelemesi yaparak, uygulamanın performansını artırır, yükü dağıtır ve yüksek erişilebilirlik sağlar.
    - Bir mikroservis mimarisinde, aynı hizmetten birden fazla örneğin çalıştırıldığı durumlarda, Load Balancer bu örnekler arasında istekleri dağıtabilir.

- Örneğin, bir web uygulaması için bir Load Balancer, gelen HTTP isteklerini arka planda çalışan farklı sunuculara yönlendirebilir ve böylece yükü dengeler.

Q8 –

Hibernate, JPA (Java Persistence API) ve Spring Data, Java tabanlı uygulamalarda veritabanı etkileşimi için kullanılan popüler çerçevelerdir;

1. **Hibernate**:
    - Hibernate, Java uygulamaları için nesne ilişkisel eşleme (ORM) çerçevesidir. Nesneleri ilişkisel veritabanı tablolarına eşleme ve veritabanı işlemlerini gerçekleştirme sürecini otomatikleştirir.
    - Örnek: Bir kullanıcı nesnesi oluşturduğunuzu ve bu nesneyi bir veritabanına kaydetmek istediğinizi düşünün. Hibernate kullanarak bu işlemi gerçekleştirebilirsiniz:

```
// Kullanıcı nesnesi oluştur
User user = new User();
user.setName("John Doe");
user.setEmail("john@example.com");

// Hibernate kullanarak kullanıcıyı veritabanına kaydet
Session session = sessionFactory.openSession();
session.beginTransaction();
session.save(user);
session.getTransaction().commit();
session.close();
```

2. **JPA (Java Persistence API)**:
    - JPA, Java ortamında nesne ilişkisel eşlemeyi (ORM) standartlaştıran bir API'dir. JPA, Hibernate gibi ORM çerçevelerinin altında yatan standart bir arayüzdür.

      Örnek:

```
// JPA EntityManager kullanarak kullanıcıyı veritabanına kaydet
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
entityManager.persist(user);
entityManager.getTransaction().commit();
entityManager.close();
```

Spring Data:

- Spring Data, Spring Framework içinde bulunan bir alt projedir ve veritabanı işlemlerini kolaylaştırmak için çeşitli modüller sunar.
- Spring Data JPA, JPA üzerine bir katman ekler ve JPA tabanlı veritabanı etkileşimini kolaylaştırır. CRUD işlemleri için standart yöntemler sağlar.

- Örnek:

```
// Spring Data JPA kullanarak kullanıcıyı kaydet
userRepository.save(user);
```