DefineX Java Spring Boot Bootcamp
HW-2

İlker KUŞ

İÇİNDEKİLER

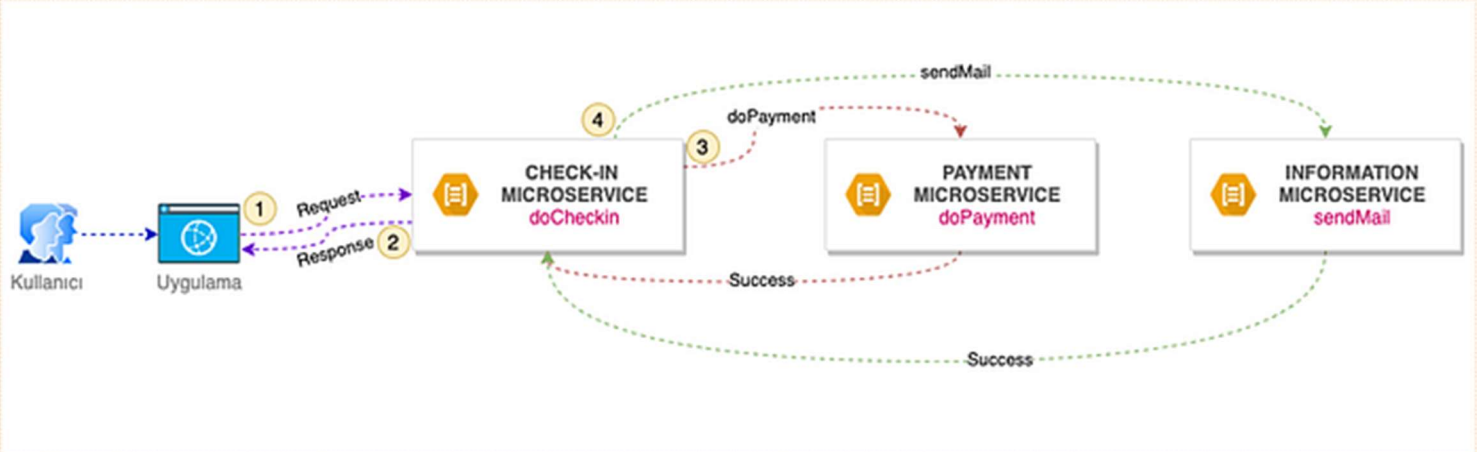
1. Senkron ve Asenkron iletişim nedir örneklerle açıklayın? (10 PUAN) 3
    1) Senkron (Eş Zamanlı) İletişim: 3
    2) Asenkron (Eş Zamansız) İletişim: 4
2. RabbitMQ ve Kafka arasındaki farkları araştırın? (10 PUAN) 5
    1) RabbitMQ: 5
    2) Apache Kafka: 6
3. Docker ve Virtual Machine nedir? (5 PUAN) 8
    1) Docker: 8
    2) Virtual Machine (VM): 8
4. Docker komutlarını örneklerle açıklayın. (5 PUAN) 11
5. Microservice ve Monotlith mimarilerini kıyaslayın. (15 PUAN) 12
    1) Monolith Mimari: 12
    2) Mikro Servis Mimari: 13
6. API Gateway, Service Discovery, Load Balancer kavramlarını açıklayın. (10 PUAN) 15
    1) API Gateway: 15
    2) Service Discovery: 15
    3) Load Balancer: 16
7. Hibernate, JPA, Spring Data framework'lerini örneklerle açıklayın. (10 PUAN) 17
    1) Java Persistence API (JPA): 17
    2) Hibernate: 17
    3) Spring Data: 19

1. **Senkron ve Asenkron iletişim nedir örneklerle açıklayın? (10 PUAN)**

Mimari evrimi monolitik uygulamalardan SOA'ya, SOA'dan da mikro servis mimarisine kadar geldi. Mikro servis mimarisi, servislerin olabildiğince küçük, yalnıza kendi işini yapan atomik her birinin kendi veri tabanına sahip servislere ayrılmasıyla inşa edilen dağıtık bir mimari yapıdır. Bu dağıtık servisler arasındaki iletişim ve bu iletişimin nasıl sağlanacağı önemli bir konudur. Bu iletişimi sağlamada kullanılan Senkron ve Asenkron İletişim örneklerini açıklamaya çalışacağım.

1) Senkron (Eş Zamanlı) İletişim:

Senkron (eş zamanlı) iletişim, bir işlemin bir diğerini beklediği bir iletişim tarzıdır. Pratikte, bu "sor ve yanıt bekle" modeline benzer. Yani bir işlem (ya da bir 'servis' ya da 'modül' gibi düşünebiliriz) başka bir işlemden bir şey yapmasını istediğinde, yanıt alana kadar bekler. Yanıt geldiğinde, işlem tamamlanır ve ilk işlem devam eder.



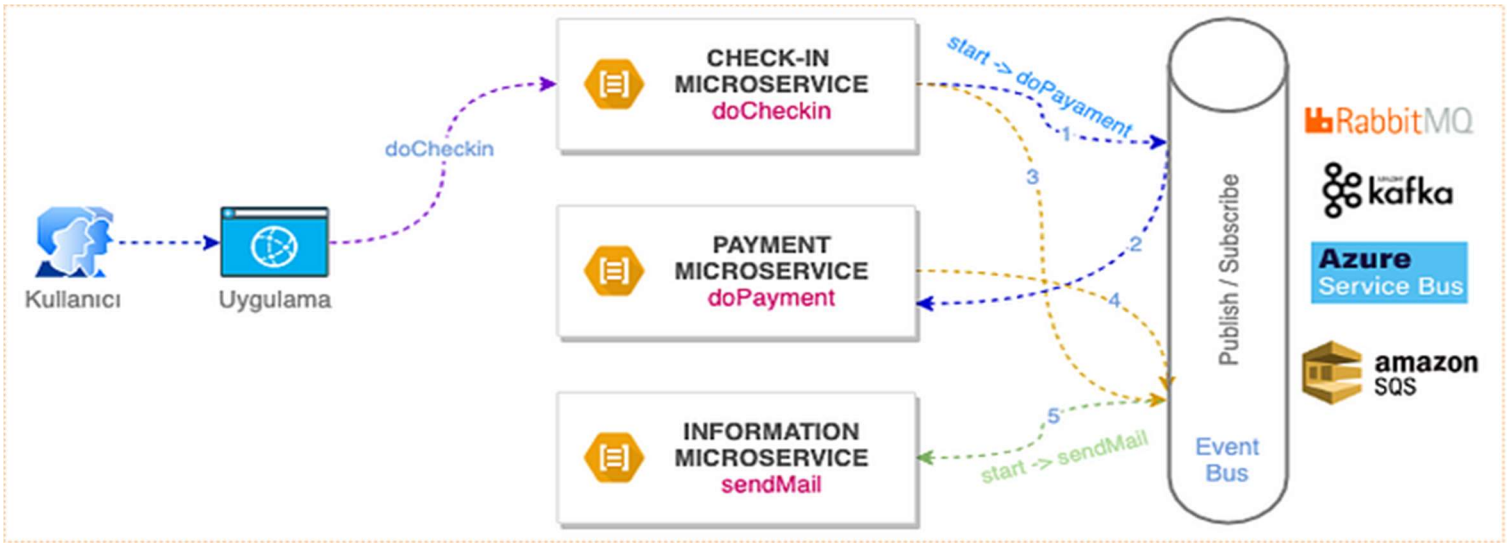
Yukarıdaki örnekte görüldüğü gibi, Senkron bir yaklaşım izlendiğinde bir mikro servis işlemlerini yaparken farklı bir servisten bir bilgi ihtiyacı olduğunda o servise istek (request) atar ve cevabını (response) bekler. Gelen cevap sonucunda ise işlemlerine devam eder.

Bir servis diğerine istekte bulunduğunda ve servis yanıtı geciktiğinde ya da hiç cevap veremediğinde istekte bulunan servis cevap beklediği için işlem başarısız olabilir. Böyle isteklerin artması da sistemde büyük bir yığılmaya ve dolayısıyla bozulmaya neden olacaktır. İşlemler başarılı bir şekilde hızlı bir yanıt alsa bile isteğin cevabını alana kadar gerekli diğer işlemler bekler ve engellenmiş olur.

Böyle bir yaklaşım için JMS (Java Message Service), RESTful API'ler veya gRPC gibi araçlar kullanılabilir.

2) Asenkron (Eş Zamansız) İletişim:

Asenkron (eş zamansız) iletişimde ise, senkron iletişimin aksine istekte bulunan servis, hizmet aldığı servisten yanıt beklemez. Böylece istekte bulunan servis hizmet aldığı serviste bağımlı hale gelmez. ***Loosely coupled*** servisler, mikro servis mimarisinde tam olarak istenilen bir durumdur.



Yukarıdaki örnekte görüldüğü gibi, Asenkron bir yaklaşımda mikro servisler birbirlerine olan bağımlılıklarından kurtulup, bir Event Bus üzerinden fırlatılan eventler ile asenkron bir iletişim sağlamıştır.

Asenkron iletişim için RabbitMQ, Apache Kafka veya Apache Pulsar gibi mesaj kuyrukları ve akış işleme platformları tercih edilebilir.

Asenkron iletişim ayrıca bire çok (**One-to-Many**) iletişim yapılabilmesine de olanak sağlar, yani bir mesaj atıldığında birden fazla servise ulaşması sağlanabilir. Senkron iletişimde ise client her bir farklı servise tek tek istekte bulunmak durumundadır.

Özet:

Senkron iletişim servisler arasındaki iletişimin kesinlikle tamamlanması gerektiği ve cevap beklendiği durumlarda kullanılmaktadır. Geri kalan tüm durumlarda Asenkron iletişim yöntemi tavsiye edilir. Asenkron iletişim için en iyi yol Event-Driven mimarisiyle birlikte Message Bus/Event Bus üzerinden iletişimin konumlandırılmasıdır.

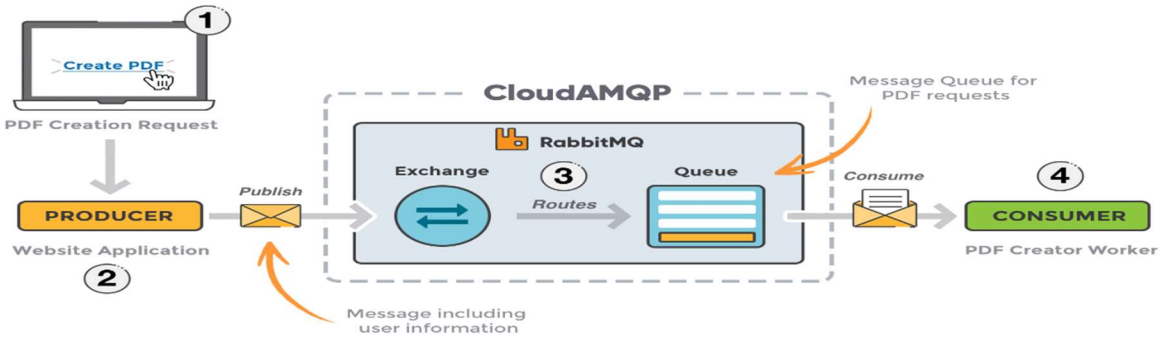
2. RabbitMQ ve Kafka arasındaki farkları araştırın? (10 PUAN)

RabbitMQ ve Kafka, akış işlemede kullanabileceğiniz mesaj kuyruğu sistemleridir. Bir mesajı gönderenin resmi mesajlaşma protokolünden alıcının resmi mesajlaşma protokolüne çeviren bir ara bilgisayar program modülüdür.

1) RabbitMQ:

RabbitMQ **mesaj kuyruğu (Message Queue)** sistemidir. Yazdığımız programımız üzerinden yapılacak **asenkron (eş zamansız)** işlemleri sıraya koyup, bunları sırayla kuyruktan çekip gerçekleyerek ilerleyen ölçeklenebilir ve performanslı bir sistemdir.

RabbitMQ, Python, Java, .NET, PHP, Ruby, JavaScript, Go, Swift ve daha fazlası dâhil olmak üzere tüm ana dilleri destekler.



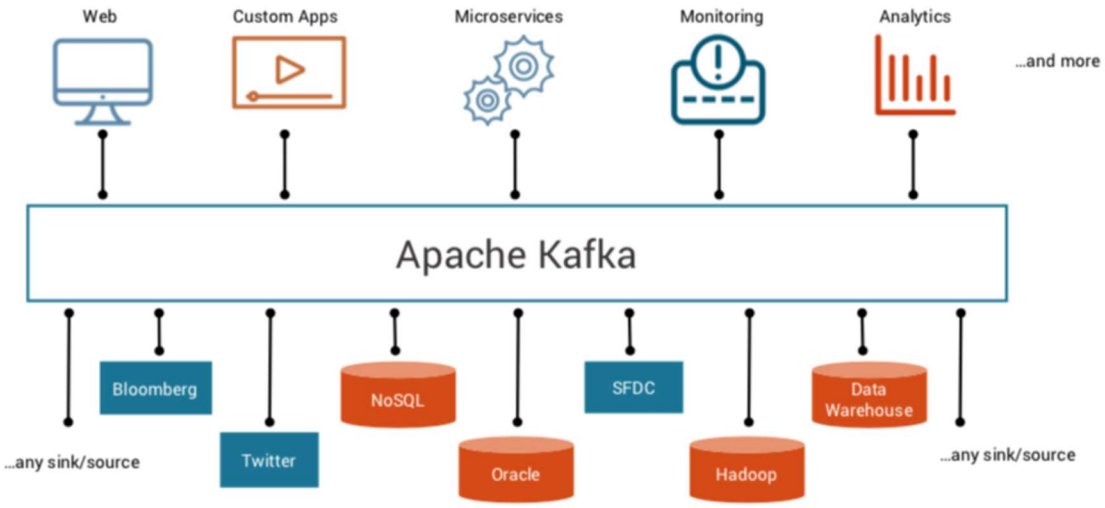
Temel RabbitMQ terimleri şunlardır:

- **Producer:** Mesajları gönderen uygulama.
- **Consumer:** Mesajları alan uygulama.
- **Queue:** Mesajları depolayan tampon.
- **Message:** Üreticiden bir tüketiciye RabbitMQ aracılığıyla gönderilen bilgiler.
- **Channel:** Bir bağlantının içindeki sanal bağlantı. Bir kuyruktaki mesajları yayınlarken veya tüketirken bunların hepsi bir kanal üzerinden yapılır.
- **Exchange:** Üreticilerden mesajlar alır ve exchange türüne göre tanımlanan kurallara göre bunları sıralara iter. Mesajları almak için bir kuyruğun en az bir exchange'e bağlanması gerekir.
- **Binding:** Binding, Queue ve exchange arasındaki bağlantıdır.
- **Routing Key:** Mesajın sıralara nasıl yönlendirileceğine karar vermek için exchange'in baktığı bir anahtardır. Routing Key'i *mesajın adresi* gibi düşünün.
- **AMQP:** Advanced Message Queuing Protocol, RabbitMQ tarafından mesajlaşma için kullanılan protokoldür.

2) Apache Kafka:

Apache Kafka, verilerin bir sistemden hızlı bir şekilde toplanıp diğer sistemlere hatasız bir şekilde transferini sağlamak için geliştirilen dağıtık bir veri akış mekanizmasıdır.

2011'de Linkedin tarafından geliştirilen Kafka daha sonra Apache çatısı altında açık kaynak bir projeye dönüştürülmüştür.



Temel Kafka terimleri şunlardır:

- **Topics:** Belirli bir veri akışını depolamak ve bu verileri yayınlamak için kullanılan kategoriyi niteler.
- **Partitions:** Topic, kendisinin bölümleri olacak şekilde ayrılmış birkaç bölümlerdir.
- **Broker:** Bir Kafka Cluster, brokers veya Kafka Brokers olarak bilinen bir veya birden fazla sunucudan oluşmaktadır. İstemcilerden gelen tüm istekleri işler ve cluster içinde çoğaltılan verileri tutar.
- **Topic Replication:** Veri kaybı durumunun yaşanmaması için bizlere bir konu çoğaltma özelliği sunar.
- **Producer:** Producer belli bir topic'e mesaj gönderen Kafka bileşenidir. Publisher (yayımcı) pozisyonundadırlar.
- **Consumer:** Consumer, belli bir topic'den mesaj okuyan Kafka bileşenidir. Subscriber (abone) pozisyonundadırlar.
- **Message Keys:** Mesaj anahtarları, mesajların belirli bir sıra ile göndermesi için kullanılır.
- **Acknowledgement:** Veri yazma işleminde producer'a bir başka onay verme seçeneğidir.

Özellikler	Apache Kafka	RabbitMQ
Mimari	Kafka, mesajlaşma sırası ile yayınlama abone yaklaşımlarını birleştiren, bölümlenmiş bir günlük modeli kullanır.	RabbitMQ bir mesajlaşma sırası kullanır.
Ölçeklenebilirlik	Kafka, bölümlerin farklı sunuculara dağıtılmasına izin vererek ölçeklenebilirlik sağlar.	Rakip tüketiciler arasında işleme ölçeğini genişletmek için sıraya giren tüketici sayısını artırır.
Mesaj saklama	Politika tabanlıdır; örneğin mesajlar bir gün boyunca saklanabilir. Kullanıcı bu saklama penceresini yapılandırabilir.	Onay tabanlıdır; yani mesajlar tüketildikçe silinir.
Birden fazla tüketici	Birden fazla tüketici aynı konuya abone olabilir çünkü Kafka aynı mesajın belirli bir zaman aralığı için tekrar oynatılmasına izin verir.	Birden fazla tüketici aynı mesajı alamaz çünkü mesajlar tüketildikçe kaldırılır.
Çoğaltma	Konular otomatik olarak çoğaltılır ancak kullanıcı, konuları çoğaltılmayacak şekilde manuel olarak yapılandırabilir.	Mesajlar otomatik olarak çoğaltılmaz ancak kullanıcı bunları çoğaltılmak üzere manuel olarak yapılandırabilir.
Mesaj sıralaması	Her tüketici, bölümlenmiş günlük mimarisi nedeniyle sırayla bilgi alır.	Mesajlar tüketicilere sıraya varış sırasına göre teslim edilir. Rakip tüketiciler varsa, her tüketici bu mesajın bir alt kümesini işleyecektir.
Protokoller	Kafka, TCP üzerinden ikili bir protokol kullanır.	Eklentiler aracılığıyla desteklenen gelişmiş mesajlaşma sırası protokolü (AMQP): MQTT, STOMP.

Peki, Kafka mı RabbitMQ mu?

Gerçek projelerde bu araçların kullanımı, öncelikle belirli iş gereksinimlerine, veri hacmine ve performans beklentilerine bağlıdır. Örneğin, büyük veri işleme projeleri genellikle Kafka veya diğer akış işleme platformlarını kullanırken, mikro servis tabanlı uygulamalar genellikle RabbitMQ gibi mesaj kuyruklarını tercih ederler.

3. **Docker ve Virtual Machine nedir? (5 PUAN)**

1) **Docker:**

Docker, uygulamaları hızla derlemeye, test etmeye ve dağıtmaya imkân tanıyan bir yazılım platformudur. Docker, yazılımları kitaplıklar, sistem araçları, kod ve çalışma zamanı dâhil olmak üzere yazılımların çalışması için gerekli her şeyi içeren **container** adlı standartlaştırılmış birimler halinde paketler. Docker'ı kullanarak her ortama hızla uygulama dağıtıp uygulamaları ölçeklendirebilir ve kodumuzu çalıştırabiliriz.

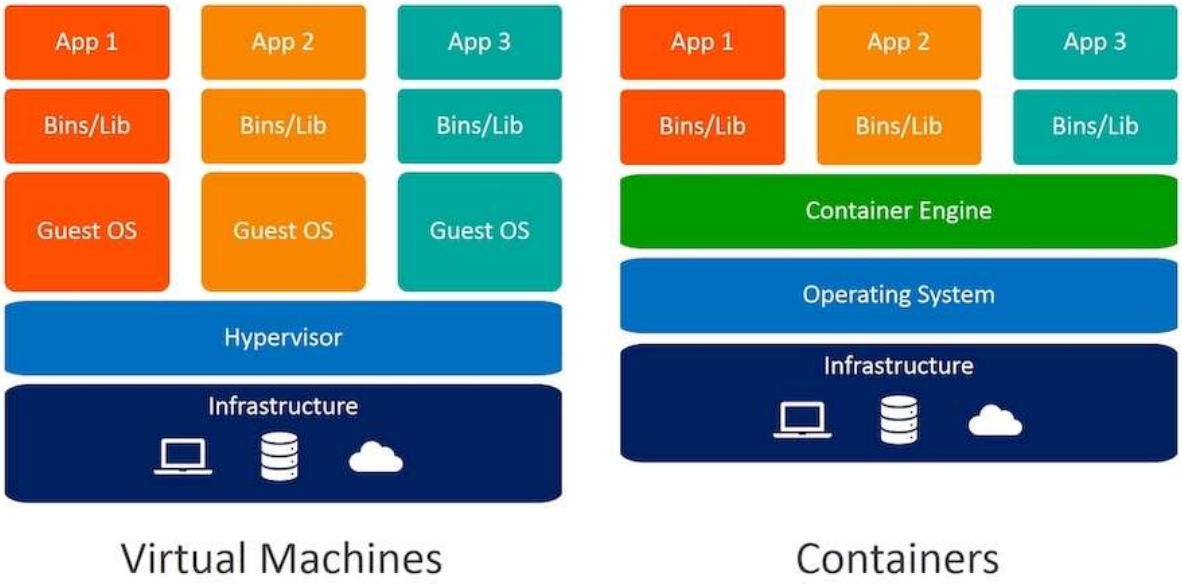
Docker Temel Bileşenleri:

- **Docker Daemon:** Hypervisor'ün dockerdaki karşılığıdır. Bütün CPU ve RAM vb gibi işletim sistemine ait işlerin yapıldığı bölümdür.
- **Docker Container:** Docker Daemon tarafından Linux çekirdeği içerisinde birbirinden izole olarak çalıştırılan process'lerin her birine verilen isimdir.
- **Docker Image:** Containerlar layer halindeki Image'lardan oluşur. Docker Image ise containerlara kurulacak ve run edilecek olan uygulamaların veya OS'lerin image dosyalarıdır. Örn: mysql, mongodb, redis, ubuntu, mariadb..
- **Docker Registry:** Imagelerin Docker Hub üzerinde depolandığı konumu işaret eder. Github'a benzer bir şekilde Docker Registry, oluşturulan imagelerin başkaları tarafından kullanılmasını sağlar.
- **Dockerfile:** Uygulama projesinde image oluşturmak için ihtiyaç duyulan config dosyasının ismi olarak tanımlanabilir. Konteyner yapılandırmasında; ağ bağlantı noktası, dil ve dosya konumu gibi birçok image oluşturulurken gerekli tüm talimatlar Dockerfile ile katmanlaştırılır.

2) **Virtual Machine (VM):**

Sanal Makine (Virtual Machine-VM), gerçek bir bilgisayar olarak işlev gören bir bilgisayar dosyasıdır. VM'ler kullanıcılar için sanal bir ortam oluşturur. Sanal makineler, bilgisayarın işletim sistemindeki bir pencerede işlem olarak çalışır. Kullanıcılar bu ortamlarda uygulamalar çalıştırabilir, veri depolayabilir ve herhangi bir bilgisayarda yapılabilecek herhangi bir eylemi gerçekleştirebilir.

Docker vs Virtual Machine



Virtual Machines

Containers

VM (Virtual Machine)

- OS : **Tam işletim sistemi**
- İzolasyon : **Yüksek**
- Çalışır hale gelmesi : **Dakikalar**
- Versiyonlama : **Yok**
- Kolay paylaşılabilirlik : **Düşük**

Docker

OS : **Küçültülmüş işletim sistemi imajı**

İzolasyon : **Daha Düşük**

Çalışır hale gelmesi : **Saniyeler**

Versiyonlama : **Yüksek**

Kolay paylaşılabilirlik : **Yüksek**

Peki, Mikroservis Mimarisinde Docker Neden Önemlidir?

Mikroserviceleri küçük uygulamalara benzediğinden, farklı ortamlar sağlamak için mikroserviceleri kendi VM instancelarına dağıtmamız gerekir. Tahmin edebileceğiniz gibi, bir sanal makinenin tamamını uygulamanın yalnızca küçük bir bölümünü dağıtmaya ayırmak en etkili yöntem değildir. Bununla birlikte, Docker ile performans ek yükünü azaltmak ve aynı sunucuya binlerce mikroservice dağıtmak mümkündür. Çünkü Docker conteinerları sanal makinelere göre çok daha az sistem kaynağı gerektirir.

Farklar	Virtual Machine	Docker
İşletim Sistemi	○ Ana bilgisayar çekirdeğini paylaşmaz. ○ Her uygulamanın eksiksiz bir işletim sistemine veya Hypervisor'e ihtiyacı vardır.	○ Docker, container'lar herhangi bir işletim sisteminde uygulama bazında kullanılabilir. ○ Docker'da, container'lar ana işletim sistemi çekirdeğini paylaşırlar. ○ Bu teknolojide, birden çok iş yükü tek bir işletim sisteminde çalışabilir.
Performans	○ Bir VM sistem üzerine işletim sistemi kurulduğu için daha fazla kaynağın kullanılmasına neden olur. ○ İşletim sisteminin yüklenmesi ve uygulamanın hazırlanması Docker'a göre daha yavaş kalır.	○ Docker container'ları aynı işletim sistemini kullandıkları için daha hızlı başlatılabilir. Başka bir deyişle başlatma süresi daha kısa olur.
Taşınabilirlik	○ Taşınabilirlik sorunları vardır. ○ VM'lerin merkezi bir hub'ı yoktur ve verileri depolamak için daha fazla bellek alanı gerektirir. ○ Dosyaları aktarırken, VM işletim sisteminin ve bağımlılıklarının bir kopyasına sahip olması gerekir. Dosya boyutunun artması aktarımı daha yavaşlatır.	○ Docker container ile kullanıcılar bir uygulama oluşturabilir ve bunu bir container görüntüsünde (image) depolayabilir. Ardından, herhangi bir ana bilgisayar ortamında çalıştırabilir. ○ Docker container sanal makinelerden daha az yer kaplar. Bu sayede bilgisayarınızın dosya sistemindeki dosyaları aktarma işlemi daha kolay şekilde olur.
Hız	○ İşletim sistemi de ayrıca başlatıldığı için uygulamanın ayağa kalkması zaman alır. ○ Tek bir uygulamayı dağıtmak için tüm işletim sisteminin kopyalanması ve taşınması gerekir.	○ Docker container'ındaki uygulama, işletim sistemi zaten hazır ve çalışır durumda olduğundan gecikme olmadan başlar. ○ Container temel olarak bir uygulamanın dağıtım sürecinde zamandan tasarruf etmek için tasarlanmıştır.

4. Docker komutlarını örneklerle açıklayın. (5 PUAN)

- **Docker Login:** Docker login komutu ile Docker kayıt defterine giriş yapabilirsiniz, burada her türlü Docker imajını çekebiliriz.

```
docker login
```

- **Docker Pull:** Docker pull komutu ile Docker registr'dan bir imaj çekebilirsiniz.

```
docker pull name-of-the-image
docker pull mysql
```

- **Docker Search:** Belirli bir image arıyorsanız ancak tam adından emin değilseniz Docker Search komutunu kullanabilirsiniz.

```
docker search rabbitmq
```

- **Docker Images:** Birkaç image çektikten sonra, Docker Images komutuyla mevcut olanların bir listesini alabiliriz. Image'ler ile ilgili detaylı bir liste çıkacaktır.

```
docker images
```

- **Docker RMI:** Docker RMI komutu ile kullanmadığınız Docker imagelerini silebilirsiniz. Buda Docker içerisinde size bir alan açacaktır.

```
docker rmi IMAGE-ID
docker rmi b0566df7e458
```

- **Docker Run:** İndirdiğiniz image'leri çalıştırmak için Docker Run komutu kullanılır.

```
docker run IMAGE-ID
docker rmi b0566df7e458
```

- **Docker PS:** Çalışan containerları listelemek için Docker PS komutunu çalıştırabilirsiniz.

- **Docker Stop / Start:** Bir konteyneri durdurmak için Docker Stop komutunu çalıştırabilirsiniz.Yeniden başlatmak için Docker Start komutu kullanılır.

```
docker stop CONTAINER-ID
docker start CONTAINER-ID
```

- **Docker RM:** Belirli bir container silmek için önce kabı durdurmanız ve ardından çalıştırmanız gerekir. Image da silinmek istenirse alttaki komut kullanılır.

```
docker rm CONTAINER_ID
docker rmi CONTAINER_ID
```

5. Microservice ve Monotlith mimarilerini kıyaslayın. (15 PUAN)

1) Monolith Mimari:

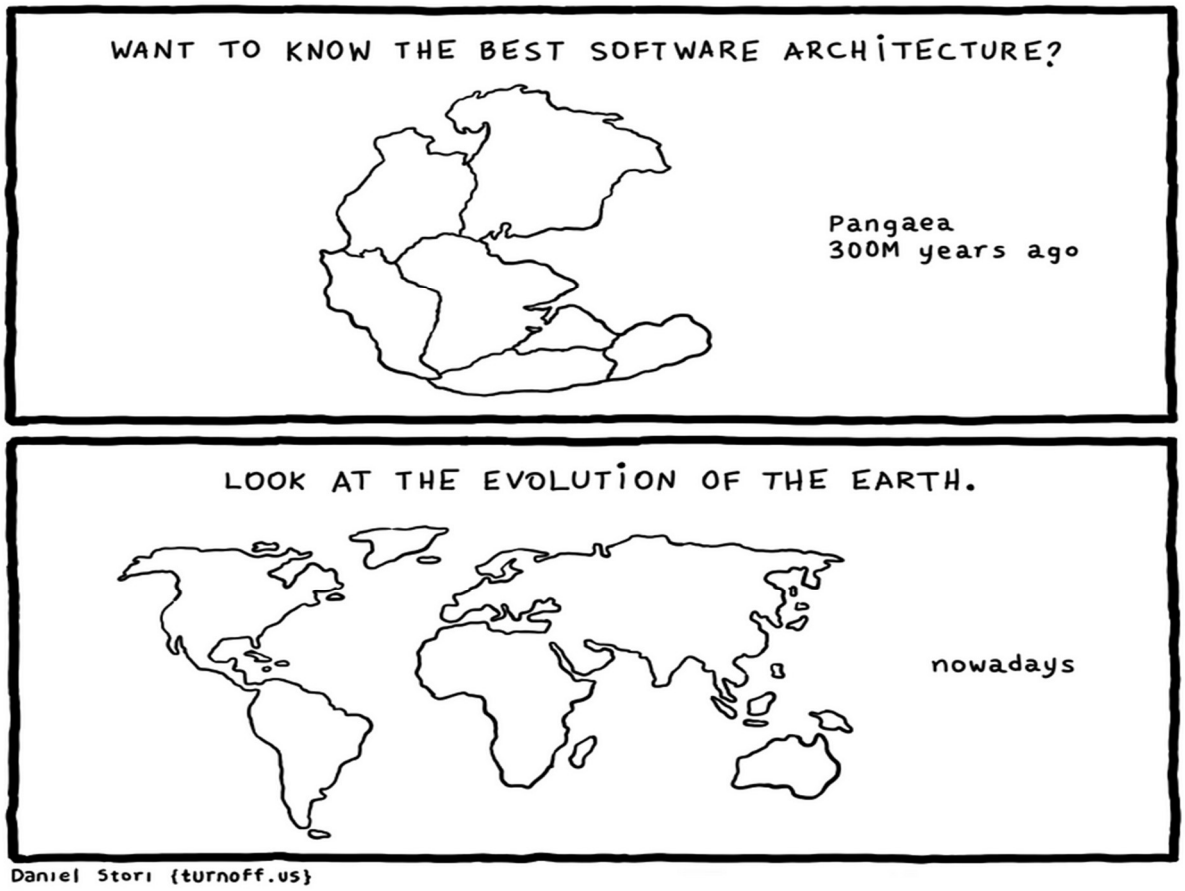
Monolithic mimari yazılımın **self-contained** (kendi kendine yeten) olarak tasarlanması anlamına gelmektedir. Bir standart doğrultusunda "tek bir parça" olarak oluşması da diyebiliriz. Bu mimarideki component'ler **loosely coupled** olmasından ziyade, **interdependent** olarak tasarlanmaktadır.

Avantajlar	Dezavantajlar
• Yönetilebilirlik ve monitoring kolaydır.	• Uygulama büyüdükçe yeni özellik geliştirilmesi ve mevcut kodun bakımı zorlaşır.
• Küçük çaplı projeler için geliştirilmesi ve bakımı kolaydır.	• Projede çalışan ekip sayısının ve çalışan sayısının artması ile geliştirme ve bakım daha güç hale gelir.
• Hızlı bir şekilde uygulama geliştirilebilir.	• Birbirlerine olan bağımlılıklarından dolayı, bir fonksiyonalitede yapılan değişiklik diğer yerleri etkileyebilir.
• Fonksiyonalitelerin birlikte çalışabilirliği tutarlıdır.	• Spesifik bir fonksiyonaliteyi scale edebilme imkanı yoktur.
• Transaction yönetimi kolaydır.	• Versiyon yönetimi zorlaşır.
	• Uygulamada aynı programlama dili ve aynı frameworklerin kullanılması gerekir.
	• Uygulamada yapılan küçük bir değişiklikte bile bütün uygulamanın deploy olması gerekir.

2) Mikro Servis Mimari:

Birbirinden bağımsız olarak çalışan ve birbirleriyle haberleşen bireysel servislerdir. Her servis kendisine ait olan iş mantığını yürütür ve diğer servislerin iş mantığı ile ilgilenmez. Plansız bir şekilde genişleyen monolithic uygulamanın hantallığını, karmaşıklığını azaltan ve yönetimini kolaylaştıran bir mimaridir.

Avantajlar	Dezavantajlar
• 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.	• Birden fazla servis ve birden fazla veri tabanı olduğu için transaction yönetimi zorlaşacaktır.
• Her servis birbirinden bağımsız ve sadece kendi iş mantıklarını bulundurduğu için servisin code base'i oldukça sade olacaktır.	• Bu servislerin yönetilebilirliği ve monitoring işlemi zorlaşacaktır.
• Ekipler daha verimli ve hızlı bir şekilde çalışabilir. Projeye yeni başlayan arkadaşlar code base de kaybolmadan kolay bir şekilde adapte olabilir.	• Servisler gereğinden fazla bağımsız olursa yönetimi zorlaşacaktır. Örneğin user ile alaklı CRUD işlemler yapan user-service adında bir servisiniz olduğunu varsayalım. Eğer siz gaza gelip user-service'ın her bir fonksiyonalitesini (create, update, delete, get… vb) ayrı servislere ayırmak isterseniz bunun yönetimi çok daha zorlaşacaktır.
• 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.	



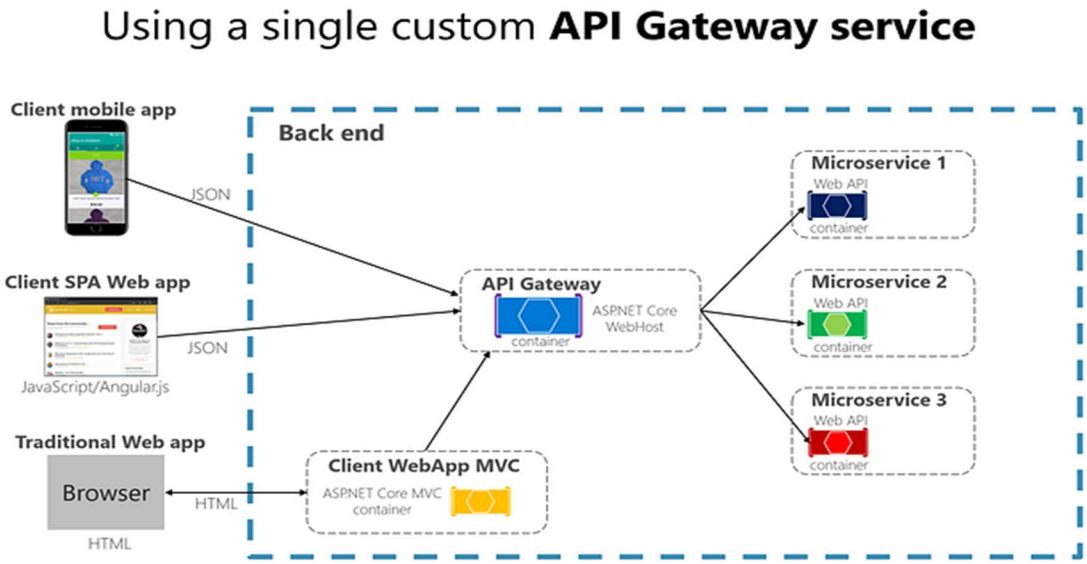
Peki, hangi mimari tercih edilmeli?

Hangi mimarinin tercih edilmesi gerektiği, projenin özel gereksinimlerine ve koşullarına bağlıdır. Küçük ölçekli projeler için monolithic mimari basitlik sağlayabilirken, büyük ölçekli ve ölçeklenebilir sistemler için mikro servis mimarisi daha uygun olabilir. En iyi yaklaşım, projenin gereksinimlerini ve gelecekteki büyüme planlarını dikkate alarak karar vermektir.

6. API Gateway, Service Discovery, Load Balancer kavramlarını açıklayın. (10 PUAN)

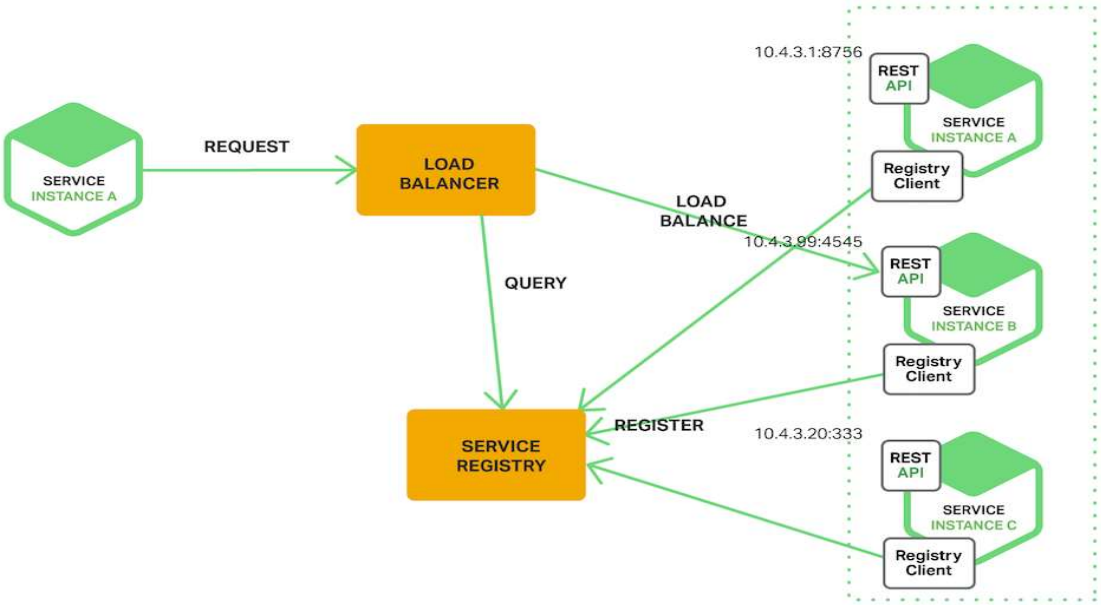
1) API Gateway:

API Gateway, bir sistem içinde bulunan farklı uygulamalar arasındaki iletişimi kolaylaştırmak ve yönetmek için kullanılan ara katman görevi gören bir yazılımdır. Bir uygulamanın arkasındaki hizmetleri, sistemleri veya mikro servisleri dış dünyaya açmak için kullanılır.



2) Service Discovery:

Servis Discovery, dağıtık sistemlerdeki servislerin bulunmasını ve birbirleriyle iletişim kurmasını sağlayan bir kavramdır. Özellikle bulut tabanlı ve mikro servis mimarilerinde yaygın olarak kullanılır. Servislerin IP adreslerinin ve bağlantı noktalarının dinamik olarak değişebileceği bir ortamda, Servis Discovery, bir servisin konumunu ve durumunu belirlemek için kullanılır. Bu, servislerin otomatik olarak yük dengeleyicilere (Load Balancer) kaydedilmesi ve güncellenmesi anlamına gelir.



3) Load Balancer:

Yük Dengeleyici, bir ağ veya sunucu kaynağının yükünü dengeleyen bir sistemdir. Özellikle yüksek trafik alanlarında ve büyük ölçekli uygulamalarda kullanılır. Yük dengeleyiciler, gelen istekleri alır ve belirli bir algoritma kullanarak bu istekleri birden fazla sunucuya veya kaynağa yönlendirir. Bu, sistemdeki yükü eşit şekilde dağıtarak performansı artırır, hizmet kesintilerini azaltır ve yüksek erişilebilirlik sağlar. Yük dengeleyiciler, karmaşık trafik yönetimi, otomatik ölçeklendirme, servis yedekleme ve izleme gibi özellikleri destekleyebilir.

Peki, Servis Discovery ile API Gateway arasındaki fark nedir?

Genellikle servisler arası iletişim karmaşıklığına, cephe görevi gören API Gateway yazılımlarıyla çözüm getirmeye çalışırız lakin **API Gateway'ler uç düzeyde clienttan servise(client-to-service)** trafiği yönettiklerinden dolayı servisler arası iletişim için yeterli bir model sunamadıklarından dolayı yetersiz gelmektedirler.

Service Discovery, service-to-service haberleşme süreçlerinde uygulamaların birbirlerinin network adreslerine metinsel değerler karşılığında ulaşmasını sağlayan basit bir model ortaya koymaktadır.

Load Balancer, trafik yönetimi ve yükü dengeleme görevini üstlenir. Service Discovery ya da API Gateway'ler ile birlikte kullanılır.

7. Hibernate, JPA, Spring Data framework'lerini örneklerle açıklayın. (10 PUAN)

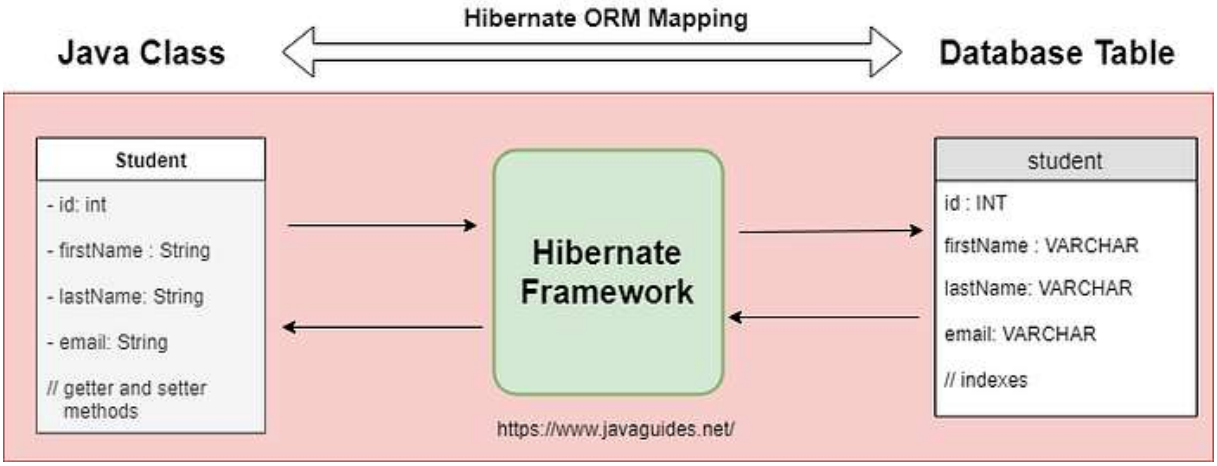
1) Java Persistence API (JPA):

JPA, Java objeleri ve ilişkisel (relational) database arasında bilgi aktarımı için kullanılan bir standarttır. JPA bu ikisi arasında bir köprü görevi görür. ORM araçları bilgiler konusunda bir devamlılık sağlamak için JPA kullanır. JPA aracılığıyla farklı ORM araçları ufak birkaç ayarlama dışında yazılan kod değiştirilmeden kullanılabilir.

```java
import javax.persistence.*;

@Entity
@Table(name = "users")
public class User {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

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

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

  public User() {}

  public User(String name, String email) {
    this.name = name;
    this.email = email;
  }

  // getters and setters

  @Override
  public String toString() {
    return id + ": " + name + " (" + email + ")";
  }
}
```

2) Hibernate:

Hibernate, nesne-tablo eşleştirmesi (Object-Relational Mapping - ORM) sağlayan bir çerçevedir ve ilişkisel veritabanlarıyla etkileşim için yüksek seviyeli bir API sunar. Java nesnelerini veritabanı tablolarına eşler ve düşük seviyeli SQL kodu yazmak yerine yüksek seviyeli bir API kullanarak veritabanı işlemleri gerçekleştirmenizi sağlar.



```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class CRUDExample {

  // Set up the database connection
  private static final String DB_URL = "jdbc:mysql://localhost/mydatabase";
  private static final String DB_USER = "myuser";
  private static final String DB_PASSWORD = "mypassword";

  public static void main(String[] args) {
    try (Connection conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD
      // Create a new record
      String insertQuery = "INSERT INTO users (name, email) VALUES (?, ?)";
      PreparedStatement pstmt = conn.prepareStatement(insertQuery);
      pstmt.setString(1, "John Doe");
      pstmt.setString(2, "johndoe@example.com");
      int rowsAffected = pstmt.executeUpdate();
      System.out.println("Inserted " + rowsAffected + " rows.");

      // Retrieve a record
      String selectQuery = "SELECT * FROM users WHERE id = ?";
      pstmt = conn.prepareStatement(selectQuery);
      pstmt.setInt(1, 1);
      ResultSet rs = pstmt.executeQuery();
      while (rs.next()) {
        int id = rs.getInt("id");
        String name = rs.getString("name");
        String email = rs.getString("email");
        System.out.println(id + ": " + name + " (" + email + ")");
      }

      // Update a record
      String updateQuery = "UPDATE users SET email = ? WHERE id = ?";
      pstmt = conn.prepareStatement(updateQuery);
      pstmt.setString(1, "newemail@example.com");
      pstmt.setInt(2, 1);
      rowsAffected = pstmt.executeUpdate();
      System.out.println("Updated " + rowsAffected + " rows.");

      // Delete a record
      String deleteQuery = "DELETE FROM users WHERE id = ?";
      pstmt = conn.prepareStatement(deleteQuery);
      pstmt.setInt(1, 1);
      rowsAffected = pstmt.executeUpdate();
      System.out.println("Deleted " + rowsAffected + " rows.");
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Hibernatesiz

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class CRUDExample {

  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenc
    EntityManager em = emf.createEntityManager();
    EntityTransaction tx = em.getTransaction();

    try {
      tx.begin();

      // Create a new record
      User user = new User("John Doe", "johndoe@example.com");
      em.persist(user);

      // Retrieve a record
      user = em.find(User.class, 1L);
      System.out.println(user);

      // Update a record
      user.setEmail("newemail@example.com");
      tx.commit();
      System.out.println(user);

      // Delete a record
      em.remove(user);
      tx.commit();
    } catch (Exception e) {
      if (tx.isActive()) {
        tx.rollback();
      }
      e.printStackTrace();
    } finally {
      em.close();
      emf.close();
    }
  }
}
```

Hibernateli

3) Spring Data:

Spring Data, JPA ile çalışmayı daha yüksek seviyede ve kullanımı daha kolay bir API sağlayan Spring Framework'ün bir parçasıdır. Spring Data JPA, JPA kullanarak bir veri tabanıyla etkileşim için gereken tekrarlayan kod miktarını azaltır ve JPA varlıkları üzerinde CRUD işlemlerini gerçekleştirmek için bir dizi yöntem sunan bir depo soyutlaması sağlar.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CRUDExample implements CommandLineRunner {

  @Autowired
  private UserRepository userRepository;

  public static void main(String[] args) {
    SpringApplication.run(CRUDExample.class, args);
  }

  @Override
  public void run(String... args) {
    // Create a new record
    User user = new User("John Doe", "johndoe@example.com");
    userRepository.save(user);

    // Retrieve a record
    user = userRepository.findById(1L).orElse(null);
    System.out.println(user);

    // Update a record
    user.setEmail("newemail@example.com");
    userRepository.save(user);
    System.out.println(user);

    // Delete a record
    userRepository.delete(user);
  }
}
```

Özetle, Hibernate, veri tabanlarıyla etkileşim için yüksek seviyeli bir API sağlayan bir ORM çerçevesidir, JPA, Hibernate gibi ORM çerçeveleri için ortak bir API tanımlayan bir spesifikasyondur ve Spring Data, JPA ile çalışmayı daha yüksek seviyede ve kullanımı daha kolay hale getiren Spring Framework'ün bir parçasıdır.

KAYNAKÇA

- https://yteblog.bilgem.tubitak.gov.tr/mikroservis-mimarisi-senkron-iletisim
- https://yteblog.bilgem.tubitak.gov.tr/mikroservis-mimarisi-iletisime-giris
- https://medium.com/@mesutpiskin/mikroservis-yakla%C5%9F%C4%B1m%C4%B1nda-servisler-aras%C4%B1-i%CC%87leti%C5%9Fim-mimarileri-27271375f5cd
- https://gokhana.medium.com/microservice-mimarisinde-servisler-aras%C4%B1-i%CC%87leti%C5%9Fim-ve-event-driven-mimari-e02cdf87e5fe
- https://medium.com/@mehmetbaz/apache-kafka-vs-rabbitmq-82128c579e66
- https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html?gad_source=1&gclid=CjwKCAjw7-SvBhB6EiwAwYdCAcIJt_2Sd7bJ5m6vPypUaZpuqRCGHVXjflJk5eZgyzeTMZrYCjPBqhoCQMAQAvD_BwE
- https://aws.amazon.com/tr/compare/the-difference-between-rabbitmq-and-kafka/
- https://aws.amazon.com/tr/compare/the-difference-between-docker-vm/
- https://erdincuzun.com/docker/01-01-docker-vs-vm/
- https://medium.com/batech/docker-nedir-docker-kavramlar%C4%B1-avantajlar%C4%B1-901b37742ee0
- https://www.techcareer.net/blog/docker-nedir-nasil-calisir
- https://www.argenova.com.tr/basit-docker-komutlari
- https://medium.com/kodgemisi/docker-etkili-kullanmak-icin-komutlar-93faf9329142
- https://aws.amazon.com/tr/compare/the-difference-between-monolithic-and-microservices-architecture/
- https://medium.com/@cengizhanozcan92/monolithic-ve-microservice-mimarisi-97874714fbf
- https://www.linkedin.com/pulse/api-gateway-ve-load-balancer-%C5%9Fenol-din%C3%A7/?originalSubdomain=tr
- https://medium.com/i%CC%87yi-programlama/microservice-mimarilerinde-service-discovery-7a6ebceb1b2a
- https://medium.com/@burakkocakeu/jpa-hibernate-and-spring-data-jpa-efa71feb82ac