1.Senkron ve Asenkron iletişim nedir örneklerle açıklayın?

Senkron İletişim:

Senkron iletişimde, gönderici ve alıcı arasındaki veri transferi, belirli bir zaman diliminde gerçekleşir. Gönderici ve alıcı arasında bir saat kaynağı paylaşılır ve veri, bu ortak saat çerçevesinde gönderilir. Senkron iletişimde, veri paketleri birbirlerine düzenli aralıklarla gönderilir ve alınır. Senkronizasyon, gönderici ve alıcı arasındaki doğru iletişimi sağlamak için kritik öneme sahiptir.

Örnekler:

Bir LAN (Yerel Alan Ağı) üzerindeki Ethernet iletişimi: Bilgisayarlar arasında veri aktarımı için Ethernet protokolü kullanılır. Gönderen ve alıcı, bir saat sinyali üzerinde senkronize olur ve veri paketleri belirli bir zaman diliminde iletilir.

Telefon konuşmaları: Geleneksel telefon hatları üzerinden yapılan konuşmalarda, konuşma sinyali gönderici ve alıcı arasında senkronize bir şekilde aktarılır.

Asenkron İletişim:

Asenkron iletişimde, gönderici ve alıcı arasında sabit bir saat kaynağı yoktur ve veri paketleri, herhangi bir zaman diliminde gönderilir ve alınır. Gönderici, veriyi alıcıya gönderir ve alıcı, veriyi alırken göndericiden bağımsız bir zaman diliminde çalışır. Asenkron iletişimde, veri paketlerinin iletimi için başlangıç ve bitiş işaretleri kullanılır ve senkronizasyon daha az önemlidir.

Örnekler:

E-posta iletişimi: Bir kişi e-posta gönderdiğinde, alıcı e-postayı daha sonra alır. Gönderici ve alıcı arasında sabit bir saat kaynağı olmadığından, iletişim asenkron olarak gerçekleşir.

USB (Universal Serial Bus): Bilgisayarlar ve çeşitli cihazlar arasında veri transferi için USB kullanılır. Cihazlar, herhangi bir zamanda veri gönderebilir veya alabilirler, bu nedenle USB iletişimi asenkron olarak kabul edilir.

2. RabbitMQ ve Kafka arasındaki farkları araştırın?

Mesaj Modeli:

RabbitMQ, tipik bir mesaj kuyruğu (message queue) modeli sunar. Bu, gönderenin mesajları kuyruğa yerleştirdiği, alıcıların ise kuyruktan mesajları alıp işlediği bir yapıdır. Mesajlar genellikle bir kuyruğa yerleştirilir ve alıcılar bu kuyruktan mesajları işler.

Kafka ise bir olay günlüğü (event log) modeli sunar. Verileri bir dizi sıralı günlükte depolar. Her bir günlük, belirli bir konu (topic) altında bulunan olayların sırasını korur. Alıcılar, günlüklerden (log) okur ve istedikleri konulara abone olabilirler.

Kullanım Senaryoları:

RabbitMQ, genellikle iş akışlarını, mesajların güvenli bir şekilde iletilmesini ve işlenmesini gerektiren durumlar için tercih edilir. Örneğin, iş kuyrukları, RPC (Uzak Yöntem Çağrısı) senaryoları ve olaya dayalı işlem modelleri gibi senaryolar için kullanılabilir.

Kafka, genellikle büyük veri akışları ve gerçek zamanlı veri işleme için tercih edilir. Büyük ölçekli veri akışları, günlükleme, analiz, olay işleme ve gerçek zamanlı uygulamalar gibi senaryolar için kullanılabilir.

Ölçeklenebilirlik:

Kafka, yüksek performanslı ve ölçeklenebilir bir yapı sunar. Verileri parçalara böler ve birden çok sunucu üzerinde dağıtır, böylece büyük miktarda veri ve yüksek talepleri işleyebilir.

RabbitMQ, Kafka'ya kıyasla daha geleneksel bir mesajlaşma sistemi olarak kabul edilir ve ölçeklenebilirlik konusunda bazı sınırlamaları olabilir.

Veri Saklama Süresi:

RabbitMQ'da, mesajlar genellikle kuyruklardan alındıktan sonra silinirler. Mesajlar, genellikle alıcı tarafından işlendikten sonra hemen kuyruktan çıkarılır.

Kafka'da ise, veriler varsayılan olarak bir süre boyunca saklanır. Bu, alıcıların veriyi daha sonra işlemesi veya geriye dönük analiz için veriye erişebilmesi anlamına gelir.

3. Docker ve Virtual Machine nedir?

Docker:

Docker, uygulama ve hizmetlerin konteyner adı verilen hafif ve taşınabilir birimler içinde paketlenmesini ve çalıştırılmasını sağlayan bir konteynerleştirme platformudur. Docker, yazılım uygulamalarını ve tüm bağımlılıklarını (örneğin kütüphaneler, dosyalar, ortam değişkenleri) bir araya getirerek, taşınabilir, hızlı ve tutarlı bir şekilde çalıştırmayı mümkün kılar.

Hafif ve Hızlı: Docker konteynerleri, sanal makinelerden daha hafif ve daha hızlıdır. Çünkü Docker konteynerleri, işletim sistemi çekirdeğini paylaşır ve üzerine eklenen uygulamaları izole eder.

Taşınabilirlik: Docker konteynerleri, herhangi bir platformda (geliştirme ortamından üretim ortamına kadar) aynı şekilde çalışabilir. Bu, yazılım uygulamalarının kolayca taşınabilir ve dağıtılabilir olmasını sağlar.

Yönetim ve Otomatizasyon: Docker, uygulama dağıtımını ve yönetimini otomatikleştirmek için araçlar sunar. Docker Compose gibi araçlar, birden fazla konteyneri bir araya getirerek kompleks uygulama yapılandırmalarını yönetmeyi kolaylaştırır.

Sanal Makine (Virtual Machine - VM):

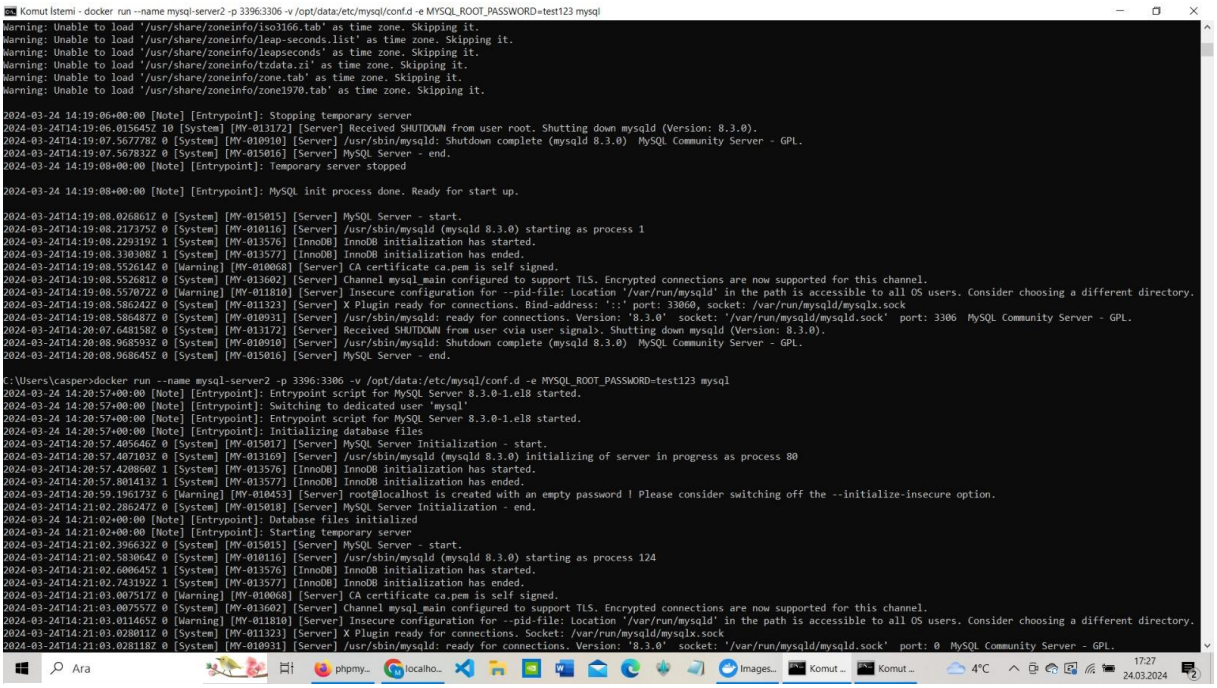
Sanal makine, fiziksel bir bilgisayar üzerinde sanal bir bilgisayar ortamı sağlayan ve bu ortamda birden çok işletim sistemi çalıştırılmasını mümkün kılan bir sanallaştırma teknolojisidir. Her bir sanal makine, kendi işletim sistemi ve bağımsız bir çevre sağlar.

Tam İzolasyon: Sanal makineler, fiziksel kaynakları paylaşan bağımsız işletim sistemleri sağlar. Her bir sanal makine, kendine ait bellek, işlemci, disk alanı ve ağ bağlantısı gibi kaynaklara sahiptir.

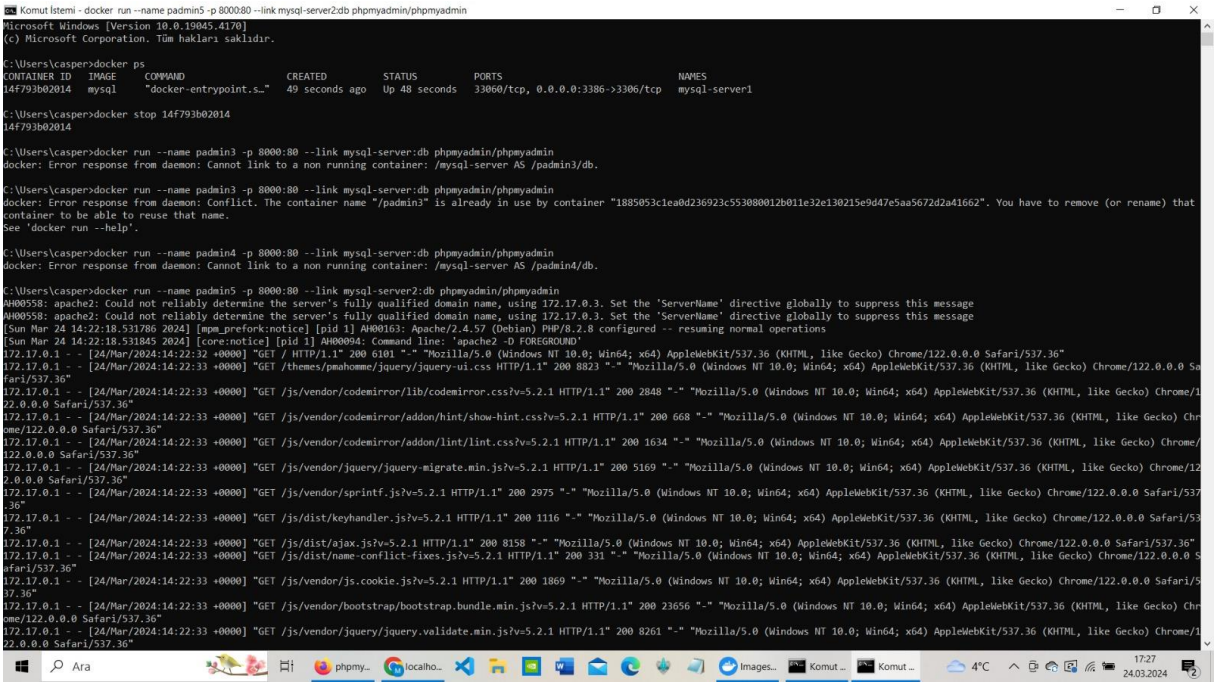
Çeşitli İşletim Sistemleri: Sanal makineler, farklı işletim sistemlerini (Windows, Linux, macOS vb.) aynı fiziksel sunucu üzerinde çalıştırmayı mümkün kılar.

Yüksek İzolasyon Maliyeti: Sanal makineler, daha fazla kaynak tüketir ve daha yüksek sistem yönetimi maliyetlerine sahiptir. Her bir sanal makine, kendi işletim sistemini barındırdığı için daha fazla bellek ve işlemci kaynağı gerektirir.

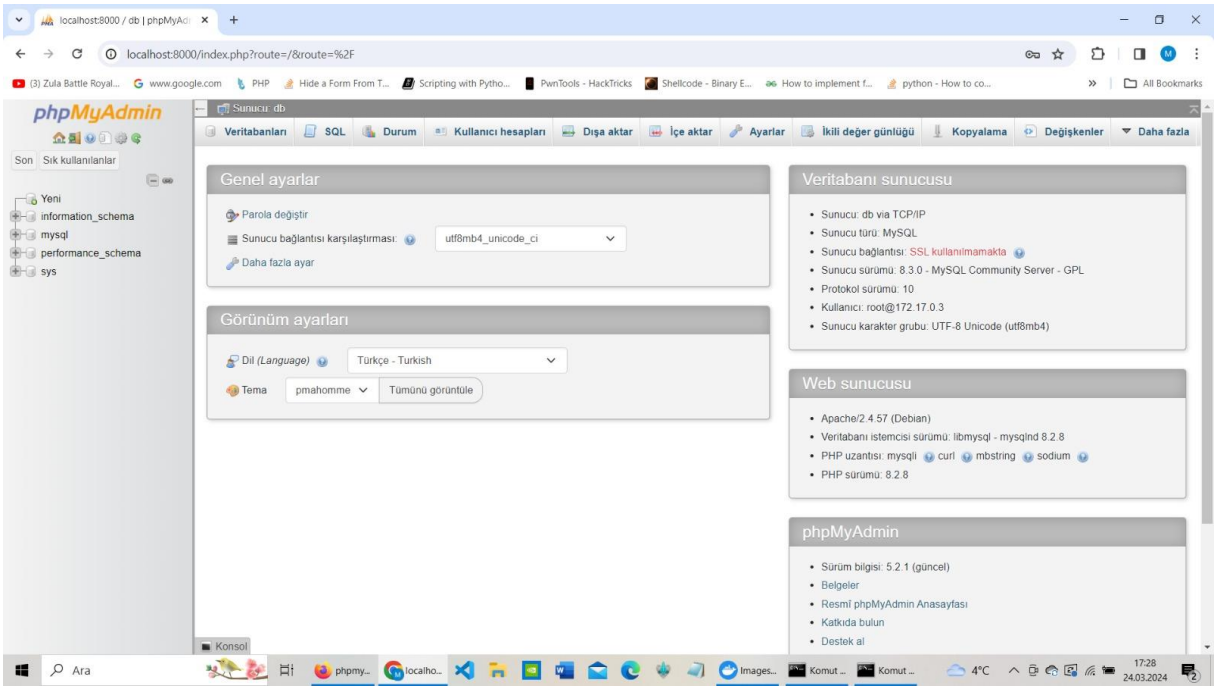
4. Docker ile RabbitMQ ve PostgreSQL ve ya MySQL kurulumu yapın?



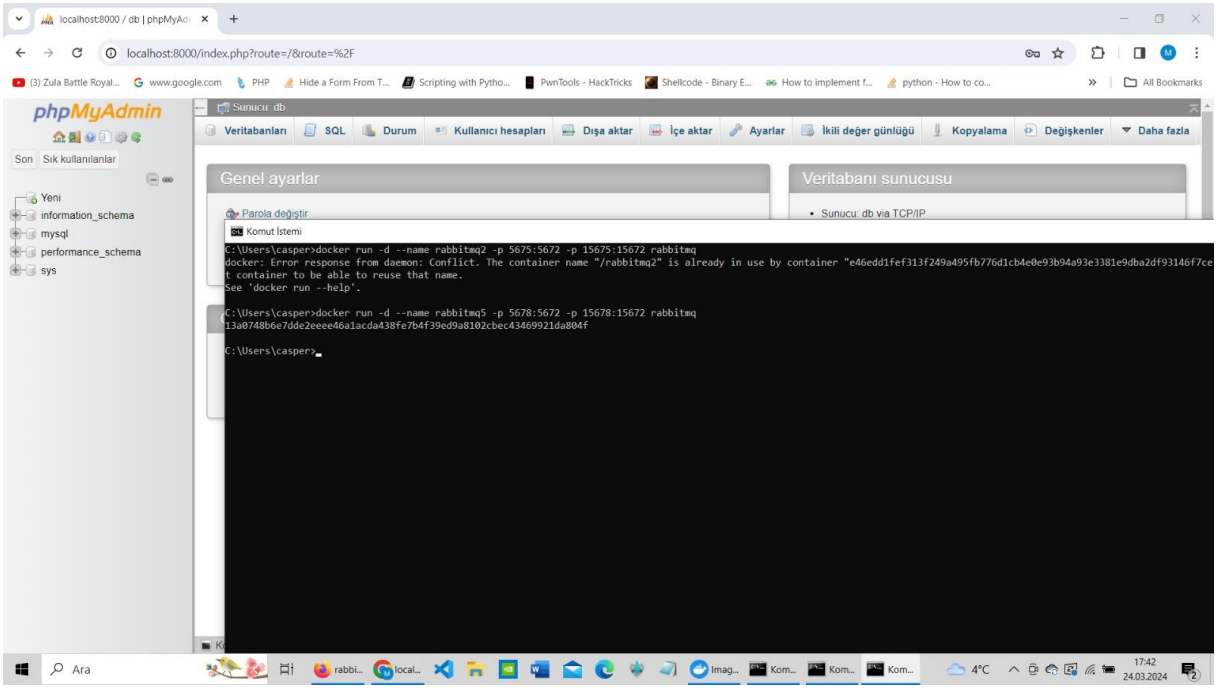
Mysql kurulum



phpMyAdmin kurulum



Localhost dan phpMyAdmin aracılığıyla MySql e bağlandım



RabbitMq kurulumu

5. Docker komutlarını örneklerle açıklayın.

docker run: Docker konteynerini çalıştırmak için kullanılır.

docker ps: Çalışan Docker konteynerlerini listelemek için kullanılır.

docker build: Docker imajlarını oluşturmak için kullanılır.

docker stop: Çalışan bir Docker konteynerini durdurmak için kullanılır.

docker rm: Bir Docker konteynerini silmek için kullanılır.

docker images: Mevcut Docker imajlarını listelemek için kullanılır.

6. Microservice ve Monotlith mimarilerini kıyaslayın.

Monolithic (Monolitik) Mimariler:

Monolithic mimariler, uygulamanın tüm bileşenlerinin tek bir büyük ve bağımlı yapıda bir araya getirildiği bir yaklaşımı ifade eder. Bir Monolitik uygulama, genellikle tek bir kod tabanı altında tüm işlevselliği barındırır. Bu yaklaşımda, birçok farklı modül veya bileşen bir araya gelir ve bu bileşenler genellikle aynı kod tabanı içinde derlenir, paketlenir ve dağıtılır.

Kolay Geliştirme ve Dağıtım: Monolitik uygulamalar, tek bir kod tabanında olduğu için geliştirme ve dağıtım genellikle daha kolaydır.

Büyük Ölçekli Değişiklikler: Uygulamanın farklı bölümlerinde yapılan değişiklikler, genellikle tüm uygulamayı yeniden derleme ve dağıtma gerekliliğini beraberinde getirir.

Ölçeklenebilirlik Zorlukları: Monolitik uygulamalar, ölçeklenebilirlik konusunda bazı zorluklar yaşayabilir. Örneğin, belirli bir bileşeni ölçeklendirmek, genellikle tüm uygulamanın ölçeklendirilmesini gerektirir.

Microservice Mimarileri:

Microservice mimarileri, uygulamanın farklı işlevselliğini birbirinden bağımsız, küçük ve ölçeklenebilir hizmetlere ayırma felsefesini benimser. Her bir hizmet, tek bir işlevselliği sağlar ve kendi geliştirme, dağıtım ve ölçeklendirme süreçlerine sahiptir. Bu yaklaşımda, uygulama bir dizi bağımsız hizmetten oluşur ve bu hizmetler genellikle farklı ekipler tarafından geliştirilir ve yönetilir.

Bağımsız Geliştirme ve Dağıtım: Her bir microservice, bağımsız olarak geliştirilebilir, dağıtılabilir ve ölçeklendirilebilir. Bu, geliştirme süreçlerini hızlandırabilir ve esnekliği artırabilir.

Küçük ve Ölçeklenebilir Hizmetler: Microservice mimarileri, belirli hizmetleri ölçeklendirmeyi ve yönetmeyi kolaylaştırır. Örneğin, yüksek trafik alanlarına sahip bir hizmeti ölçeklendirmek, sadece bu hizmeti etkiler ve diğer hizmetlere zarar vermez.

Karmaşıklık ve Yönetim: Microservice mimarileri, birden fazla hizmeti yönetmek için daha fazla karmaşıklık getirebilir. Hizmetler arasındaki bağımlılıkların ve iletişim gereksinimlerinin yönetilmesi zor olabilir.

Karşılaştırma:

Boyut ve Karmaşıklık: Monolithic uygulamalar genellikle daha basit ve daha az karmaşıktır, çünkü tüm işlevsellik tek bir kod tabanında toplanır. Microservice mimarileri ise daha modüler ve parçalıdır, bu nedenle daha fazla yönetim karmaşıklığına yol açabilirler.

Ölçeklenebilirlik: Microservice mimarileri, belirli hizmetleri ölçeklendirmeyi kolaylaştırırken, Monolithic uygulamalarda tüm uygulamayı ölçeklendirmek gerekebilir.

Bağımsız Dağıtım ve Geliştirme: Microservice mimarileri, hizmetlerin bağımsız olarak geliştirilmesini ve dağıtılmasını sağlar, bu da daha hızlı bir geliştirme süreci ve daha esnek bir dağıtım sağlar. Monolithic uygulamalarda ise tüm kod tabanı birlikte dağıtılır ve geliştirilir.

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

API Gateway:

API Gateway, bir mikro hizmet mimarisinde giriş noktası görevi gören bir ara yazılımdır. Gelen istekleri yönlendirir, filtreler, denetler ve gerekirse dönüştürür. Ayrıca birden fazla hizmetten veri toplayabilir ve birleştirerek tek bir cevap döndürebilir. API Gateway, mikro hizmetlerin arkasındaki karmaşıklığı gizleyerek, istemciler için tek bir, tutarlı bir arayüz sağlar. Güvenlik, oturum yönetimi, yetkilendirme ve kimlik doğrulama gibi ortak işlevleri de sağlayabilir.

Service Discovery:

Service Discovery, bir mikro hizmet mimarisindeki hizmetlerin dinamik olarak bulunması ve iletişim kurulması için kullanılan bir mekanizmadır. Hizmetlerin IP adresleri ve bağlantı noktaları genellikle değişkendir, bu nedenle Service Discovery, bir hizmetin konumunu (IP adresi ve bağlantı noktası) dinamik olarak belirler ve diğer hizmetlere bu bilgiyi sağlar. Böylece, hizmetler birbirleriyle etkileşime geçebilir.

Load Balancer:

Load Balancer, gelen istekleri birden fazla sunucu arasında eşit bir şekilde dağıtarak ağdaki yükü dengeleyen bir sistemdir. Bir mikro hizmet mimarisinde, belirli bir hizmetin birden fazla örneği olabilir ve bu örnekler arasında yük dengesini sağlamak önemlidir. Load Balancer, istekleri alır ve bu istekleri hedeflenen hizmet örneklerine dağıtarak yükü dengeler.

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

Hibernate:

Hibernate, Java platformu için bir ORM (Object-Relational Mapping) framework'üdür. Veritabanı tablolarını Java nesnelerine ve Java nesnelerini veritabanı tablolarına eşleştirmeyi sağlar. Bu, geliştiricilere veritabanı işlemlerini yaparken SQL yerine Java nesneleriyle çalışma imkanı sunar. Özellikle büyük ve karmaşık veritabanı yapılarıyla çalışırken geliştirme sürecini kolaylaştırır.

```
@Entity

@Table(name = "students")

public class Student {
```

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

    private String name;
    private int age;
    // Getter ve setter metotları
}
```

Yukarıdaki örnek, Hibernate ile bir JPA (Java Persistence API) varlığı tanımlar. @Entity anotasyonu, bu sınıfın bir JPA varlığı olduğunu belirtir. @Table anotasyonu, bu varlığın veritabanındaki karşılık geldiği tablonun adını belirtir. @Id ve @GeneratedValue anotasyonları, veritabanı kimlik sütununu ve otomatik artan stratejiyi tanımlar.

JPA (Java Persistence API):

JPA, Java platformu için bir ORM standardıdır. Veritabanı işlemlerini yapmak için kullanılan bir API setidir. JPA, Hibernate gibi ORM framework'leri tarafından uygulanır ve bu framework'ler JPA spesifikasyonunu takip ederek ORM yeteneklerini sağlarlar.

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

public class Main {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("example-unit");
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();

        Student student = new Student();
        student.setName("John Doe");
```

```
        student.setAge(25);
        em.persist(student);

        em.getTransaction().commit();

        em.close();
        emf.close();
    }
}
```

Yukarıdaki örnek, JPA kullanarak bir öğrenci nesnesini veritabanına kaydetmek için EntityManager aracılığıyla bir işlem gerçekleştirir.

Spring Data:

Spring Data, Spring ekosistemindeki veritabanı işlemlerini kolaylaştıran bir alt projedir. Spring Data, farklı veritabanlarıyla etkileşim sağlamak için modül tabanlı bir yapıya sahiptir. Veritabanına erişim kodunu azaltırken, temel CRUD işlemlerini ve sorgu işlemlerini basitleştirmek için özelleştirilebilirliği artırır.

```
import org.springframework.data.repository.CrudRepository;

public interface StudentRepository extends CrudRepository<Student, Long> {
}
```

Yukarıdaki örnek, Spring Data ile bir JPA repository arayüzünü temsil eder. CrudRepository interface'i, temel CRUD işlemlerini gerçekleştirmek için kullanılabilir. Student sınıfı, bu repository'nin yönettiği varlık türünü ve Long ise varlığın kimlik türünü belirtir. Bu repository, öğrenci varlıkları üzerinde CRUD işlemlerini gerçekleştirebileceğimiz yöntemler sağlar.

Mehmet Emin Meşe