TUANA EKIN ÇELIK

- Kubernetes nedir?
- Docker nedir?
- Mikro Service nedir?

2024
2025

İstanbul Gedik Üniversitesi

İçindekiler

Docker -

Kubernetes -

MİKROSERVİS

1. Giriş ve Tanımlama

1.1.Mikroservis Nedir ?

Mikroservisler temelde bir yazılım uygulamasında belirli özellik ya da fonksiyonu sağlayan, tek bir amaca hizmet eden, birbirinden bağımsız yazılım servislerdir. Bu hizmetler bağımsız olarak bakımı yapılabilir, izlenebilir ve dağıtılabilir yapıya sahip olmalıdır. **Mikroservis** tek başına tek sorumluluğu olan ve tek iş yapan sadece o işe ait işleri yürüten modüler projelerdir.

1.2.Monolitik Mimari Mikroservis Mimarisi Karşılaştırması

- Mikroserviste uygulama küçük ve bağımsız servislere bölünmüştür, her servis farklı bir iş yapar ve kendi veritabanına sahip olabilir. Monolitikte ise uygulama bir bütün halinde çalışır ve genellikle tek veritabanı kullanır.

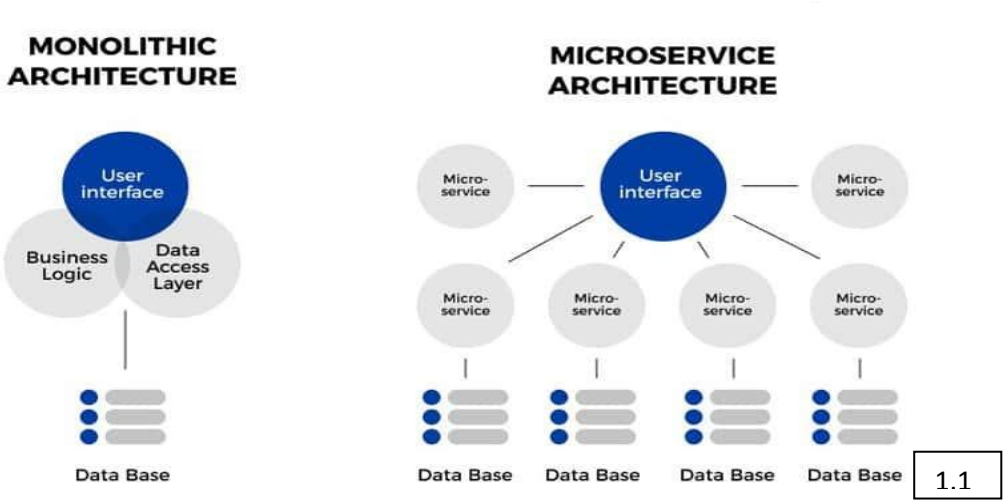
- Mikroserviste servisler bağımsız geliştirildiğinden ekip üyelerinin belirli bir işle, servisle ilgilenmesi kolaylaşır. Monolitik mimaride uygulama bütün olduğundan koordine olmak daha zordur.

- Mikroserviste daha yoğun olan servis için ekstra kaynak ayrılabilir ancak monolitikte genelde ölçeklendirme daha zordur ve tüm uygulamayı etkiler.

- Mikroserviste bir serviste oluşan hata uygulamanın çalışmasını çok etkilemez ancak monolitikte tüm projeyi etkileyebilir.

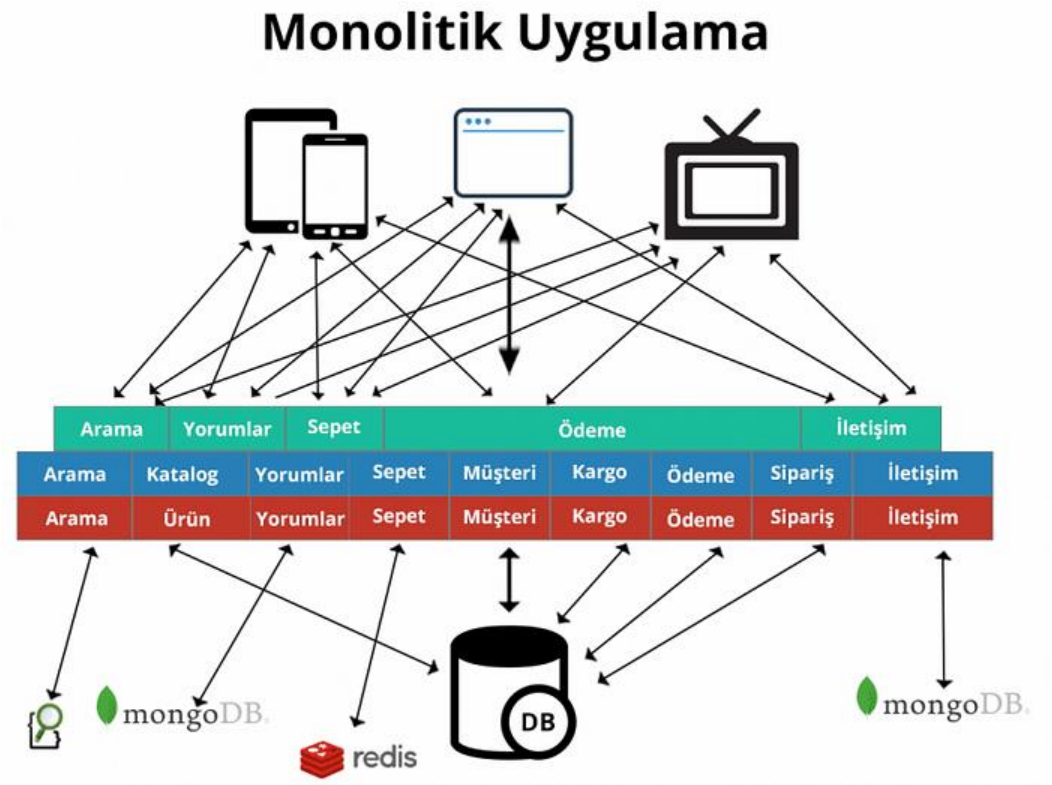
- Mikroserviste birçok teknoloji ve framework kullanılabilir, takım çalışmasında herkesin uzmanlığına göre çalışmasına imkan sağlayabilir. Monolitikte ise çoğunlukla aynı teknoloji kullanılarak proje geliştirilir.

- Mikroserviste servislerin birbirleriyle iletişimi için API tasarımının doğru yapılması önemlidir, ağdaki iletişim maliyeti yüksek olabilir. Monolitikte ise servisler aynı yerde çalıştığından aralarındaki iletişim genelde daha hızlı ve düşük maliyetlidir.



1.1

Monolitik bir uygulama, birden fazla modül içeren tek bir kod tabanına sahiptir. Modüller, fonksiyonel veya teknik özelliklerine göre ayrılmıştır. Tüm uygulamayı *build* eden tek bir derleme sistemine sahiptir. Ayrıca tek bir çalıştırılabilir veya deploy edilebilir dosyaya sahiptir. Genellikle, uygulamalar üç kısımdan oluşmakta; ***client*** kısmı yani kullanıcıya gösterilen ve kullanıcının işlemlerinin gerçekleştirildiği kısım web sayfası, mobil uygulama gibi, bir de ***sunucu*** uygulaması kısmı tüm isteklerin yönetildiği ve gerekli algoritmaların bulunduğu yapı olarak düşünülebilir ve son olarak verileri kaydettiğimiz ve depoladığımız ***database*** kısmından oluştuğu söylenebilir. Tek bir parçadan oluşan server kısmını düşündüğümüzde, sistemin tümüyle hareket ettiğini görebilirsiniz. Başlatılacağı zaman ya da durdurulacağı zaman ve hatta çöktüğü zaman uygulamanın tamamı bu durumdan etkilenir. Yani uygulama, yaşamı boyunca tek bir parça halinde hareket eder. Herhangi bir güncelleme, değişiklik ya da versiyon yükseltmesi işlemi için uygulamanın tümüne müdahale etmemiz gerekir. Buna karşılık bir mikroservis, tek işe odaklı uygulamanın işlevsel küçük bir parçasını ifade eder. Mikroservis olarak tasarlanan bir servis, ihtiyaca bağlı olarak birden fazla projede(uygulamada) veya farklı projelerde tekrar kullanılabilir. Servisler arasındaki bağımlılıklar, loose-coupled ilkesine uygun olarak en aza indirgenir. Tek sorumluluk odaklı servisler bir standarda uygun olunca, diğer servisler üzerindeki değişikliklerden etkilenmezler.

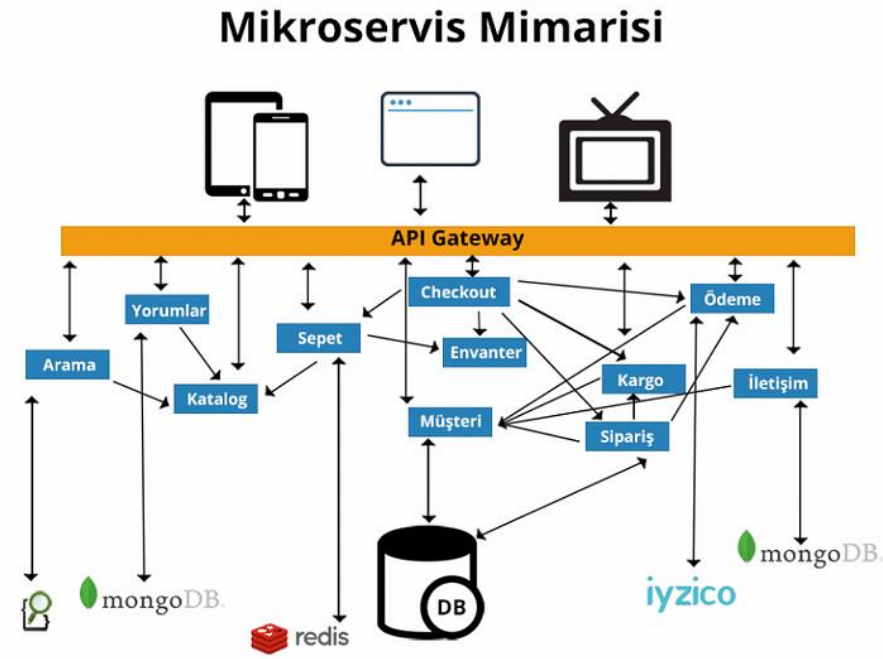


Bir e-ticaret ticaret platformunda, monolitik mimari kullanılıyorsa, tüm işlevler ve bileşenler tek bir uygulama içinde yer alır.

1. **Monolitik Uygulama:** Tüm işlevleri tek bir uygulama içinde barındıran bir monolitik yapı. Ürün yönetimi, kullanıcı yönetimi, sepet işlemleri, ödeme işlemleri ve diğer tüm işlevler aynı uygulama içinde yer alır.

2. **Tek Veri tabanı:** Tüm veriler, tek bir veri tabanında saklanır. Ürün bilgileri, kullanıcı profilleri, sepet içeriği ve sipariş verileri aynı veri tabanında tutulur.

3. **Monolitik Dağıtım:** Uygulamanın tüm bileşenleri tek bir paket olarak dağıtılır. Her bir güncelleme veya değişiklik tüm uygulamayı etkiler. Örneğin, bir ürünün güncellenmesi veya yeni bir özelliğin eklenmesi, tüm uygulamanın yeniden dağıtılmasını gerektirebilir.



Mikroservis mimarisiyle tasarlanmış bir e-ticaret platformunda aşağıdaki gibi farklı mikroservisler olabilir:

1. Ürün Hizmeti: Ürün kataloğunu yöneten ve kullanıcılara ürün bilgilerini sağlayan bir mikroservis. Bu hizmet, ürünlerin eklenmesi, güncellenmesi ve silinmesi işlemlerini gerçekleştirir.

2. Kullanıcı Hizmeti: Kullanıcıların kayıt olması, giriş yapması ve kullanıcı profillerini yönetmesiyle ilgilenen bir mikroservis. Bu hizmet, kullanıcı bilgilerini tutar, yetkilendirme işlemlerini gerçekleştirir ve kullanıcıların sipariş geçmişini yönetir.

3. Sepet Hizmeti: Kullanıcıların sepete ürün eklemesini, sepet içeriğini yönetmesini ve sipariş işlemlerini başlatmasını sağlayan bir mikroservis. Bu hizmet, kullanıcıların sepet bilgilerini tutar ve siparişlerin oluşturulması için gerekli işlemleri gerçekleştirir.

4. Ödeme Hizmeti: Kullanıcıların siparişlerini ödemesini yöneten bir mikroservis. Bu hizmet, kullanıcılardan ödeme bilgilerini alır, ödeme işlemlerini gerçekleştirir ve sipariş durumunu günceller. Her bir mikroservis, kendi bağımsız işlevselliğiyle birlikte çalışır ve ayrı bir veri tabanına sahiptir. Kullanıcıların işlem yapabilmesi için bu mikroservisler birbirleriyle iletişim kurar. Örneğin, kullanıcı bir ürünü sepete eklemek

istediğinde, kullanıcı hizmeti, sepet hizmetine bu işlemi ileterek sepete ürünün eklenmesini sağlar.

2.Mikroservislerin Avantajları

Ölçeklenebilirlik , Bağımsız dağıtım ve geliştirme , Teknoloji çeşitliliği , Hızlı ve sık güncellemeler , Ekip uyumu ve bağımsızlık

2.1.Ölçeklenebilirlik: Mikroservis mimarisi, her bir hizmetin bağımsız olarak ölçeklendirilebilmesini sağlar. Yani, yoğun taleplerin olduğu bir hizmeti yatırım yaparak ölçeklendirebilirken, diğer hizmetlerde bu ihtiyaç olmadıkça kaynak kullanımını optimize edebilirsiniz. Bu, uygulamanızın büyüdükçe daha iyi performans ve yüksek kullanılabilirlik elde etmenize yardımcı olur.

Bağımsız geliştirme ve dağıtım: Her bir mikroservis, kendi kod tabanına ve bağımlılıklarına sahiptir. Bu, farklı ekibin farklı mikroservisler üzerinde çalışabilmesini sağlar. Bir hizmetin güncellenmesi veya değiştirilmesi, diğer hizmetlere minimal etki yapar, bu da daha hızlı ve daha esnek bir geliştirme süreci sağlar. Ayrıca, hizmetleri bağımsız olarak dağıtma yeteneği, hızlı bir şekilde yeni özelliklerin veya düzeltmelerin kullanıma sunulmasını sağlar.

Teknoloji çeşitliliği: Mikroservis mimarisi, farklı teknolojileri ve dilleri kullanmanıza izin verir. Her bir hizmetin farklı bir teknoloji yığınına sahip olması mümkündür. Bu, ekip üyelerinin kendi uzmanlık alanlarında çalışabilmesini ve en iyi araçları seçebilmesini sağlar.

Esneklik: Mikroservisler, daha küçük ve bağımsız bileşenlere bölündüğü için, bir hizmetin değiştirilmesi veya yeniden yapılandırılması diğer hizmetleri etkilemez. Bu, hizmetlerin daha esnek ve değişime açık olmasını sağlar. Aynı zamanda, yeni özelliklerin veya teknolojilerin daha kolay entegre edilmesine olanak tanır.

3. Mikroservis Mimarisi

Servislerin sınırları , İletişim ve veri yönetimi , Servis keşfi ve kayıt defteri(Service discovery and registry) , Yük dengeleme ve işlem yönetimi

Servislerin Sınırları: Her mikro hizmetin kendi sınırları ve sorumlulukları vardır. Bu, hizmetin ne yapacağını ve ne yapmayacağını tanımlar. Örneğin, bir kullanıcı yönetimi hizmeti kullanıcı hesaplarını yönetirken, ödeme işlemleri hizmeti ödemeleri işler.

İletişim ve Veri Yönetimi: Mikro hizmetler genellikle HTTP RESTful API'leri veya hafif mesajlaşma protokolleri (örneğin, Kafka veya RabbitMQ) üzerinden iletişim kurarlar. Veri yönetimi açısından, her bir hizmet kendi veri tabanına sahip olabilir. Ancak, bazı durumlarda veri paylaşımı veya veri entegrasyonu gerekebilir, bu durumda dikkatli bir veri senkronizasyonu stratejisi uygulanmalıdır.

Servis Keşfi ve Kayıt Defteri: Büyük mikro hizmet sistemlerinde, hizmetlerin bulunması ve iletişim kurması için bir keşif ve kayıt defteri (service registry) kullanılabilir. Örneğin, Consul veya etcd gibi araçlar, hizmetlerin IP adreslerini ve bağlantı bilgilerini tutarak dinamik olarak hizmet keşfi sağlar.

Yük Dengeleme ve İşlem Yönetimi: Yük dengeleme, gelen istekleri farklı hizmet örnekleri arasında dağıtarak performansı artırır. Örneğin, Kubernetes veya Nginx gibi araçlar, yük dengeleme ve trafik yönetimi için kullanılabilir. İşlem yönetimi ise, belirli bir isteğin tamamlanması için gerekli mikro hizmetlerin uygun sırayla çağrılmasını ve koordinasyonunu sağlar.

Mikroservis mimarisinde bu temel unsurlar, her bir hizmetin bağımsız olarak geliştirilmesini, dağıtılmasını ve ölçeklendirilmesini sağlayarak büyük sistemlerin karmaşıklığını azaltır. Ancak, her bir unsuru etkili bir şekilde yönetmek ve uygun

araçları kullanmak önemlidir, bu da genellikle deneyim ve iyi mimari uygulamaları gerektirir.

4. Mikroservislerin Uygulama Alanları

Büyük ölçekli web uygulamaları , E-ticaret platformları , Finansal uygulamalar, Mobil uygulamalar , IoT (nesnelerin interneti) sistemleri

Büyük ölçekli web uygulamaları: Ölçeklenebilirlik, esneklik ve bağımsız dağıtım avantajları nedeniyle mikroservisler, büyük ölçekli web uygulamaları için idealdir. Her mikroservis, belirli bir işlevi veya hizmeti gerçekleştirir ve bu, uygulamanın farklı bileşenlerinin paralel olarak geliştirilmesine ve yönetilmesine olanak tanır.

E-ticaret platformları: E-ticaret sistemleri, kullanıcı deneyimini artırmak ve işletme süreçlerini optimize etmek için mikroservis mimarisinden faydalanabilir. Sipariş yönetimi, envanter yönetimi, ödeme işlemleri gibi farklı işlevler mikroservisler aracılığıyla sağlanabilir ve her bir servis bağımsız olarak ölçeklendirilebilir.

Finansal uygulamalar: Finansal uygulamalar, güvenlik, yüksek performans ve sürekli kullanılabilirlik gereksinimleri taşır. Mikroservis mimarisi, bu tür uygulamalarda işlevleri ve hizmetleri izole ederek güvenlik ve veri bütünlüğünü sağlamak için ideal bir çözümdür.

Mobil uygulamalar: Mobil uygulamalar, hafif ve hızlı yanıtlar gerektirir. Mikroservis mimarisi, mobil uygulamaların arka plandaki hizmetlerle etkileşimini optimize eder ve geliştirme süreçlerini hızlandırabilir.

IoT (nesnelerin interneti) sistemleri: IoT sistemleri, büyük miktarda veri üretir ve bu verilerin işlenmesi, depolanması ve analiz edilmesi gerekebilir. Mikroservisler, IoT cihazlarından gelen verileri işlemek için ölçeklenebilir ve esnek bir çözüm sunabilir.

Her bir uygulama alanı için mikroservis mimarisinin kullanılması, uygulamanın gereksinimlerine ve ölçeğine göre farklı avantajlar sağlar. Ancak, bu yaklaşımın

yönetimi ve koordinasyonu daha karmaşık hale getirebileceği için iyi bir planlama ve yönetim gerektirir.

5. Mikroservislerin Zorlukları ve Çözümler

Dağıtık sistemlerin karmaşıklığı , Veri bütünlüğü ve konsistensi , Hata yönetimi ve izleme , Güvenlik ve kimlik doğrulama

Dağıtık Sistem Karmaşıklığı:

Zorluklar: Mikroservislerin dağıtık doğası, ağ üzerindeki iletişim, hata yönetimi ve veri senkronizasyonu gibi konularda karmaşıklık yaratabilir.

Çözümler: Servis keşfi ve kayıt, yük dengeleme, hata toleransı ve geri alma stratejileri gibi çözümler mikroservis mimarisindeki karmaşıklığı azaltmaya yardımcı olabilir. Ayrıca, API Gateway gibi araçlar kullanarak servisler arası iletişimi kolaylaştırmak önemlidir.

Veri Bütünlüğü ve Konsistensi:

Zorluklar: Farklı servisler arasında veri konsistensinin sağlanması zor olabilir. Özellikle dağıtık işlemler ve paralel çalışma durumlarında veri bütünlüğü sorunları ortaya çıkabilir.

Çözümler: Event sourcing, CQRS (Command Query Responsibility Segregation), ve ACID (Atomicity, Consistency, Isolation, Durability) gibi veri yönetim teknikleriyle veri bütünlüğü ve konsistensi sağlanabilir. Ayrıca, eventual consistency (sonuçsal tutarlılık) gibi esnek konsistensi modelleri de düşünülebilir.

Hata Yönetimi ve İzleme:

Zorluklar: Mikroservis mimarisinde bir hata durumunda sorunun kaynağını tespit etmek ve izlemek zor olabilir. Ayrıca, hataların yayılması ve sistem genelinde etki yaratması riski bulunmaktadır.

Çözümler: Merkezi hata izleme ve loglama mekanizmaları kullanarak hataları izlemek ve analiz etmek önemlidir. Ayrıca, servisler arası iletişimde kullanılan protokoller ve güvenilirlik stratejileriyle hata yönetimi iyileştirilebilir.

Güvenlik ve Kimlik Doğrulama:

Zorluklar: Mikroservislerin sayısının artması, güvenlik açıklarını artırabilir. Ayrıca, farklı kimlik doğrulama ve yetkilendirme mekanizmalarının yönetimi de zorlaşabilir.
Çözümler: OAuth, JWT (JSON Web Token) gibi standart kimlik doğrulama yöntemleri kullanarak güvenliği artırabilirsiniz. Servisler arası iletişimde HTTPS kullanımı önemlidir. Ayrıca, güvenlik politikaları ve erişim kontrolü mekanizmaları (örneğin, API Gateway üzerinde) kurallar ve denetimler sağlamak için kullanılabilir.

Mikroservislerin bu zorluklarını ele almak için belirli senaryolar ve iş gereksinimlerine göre uygun çözümleri seçmek önemlidir. Her bir zorluğun üzerine giderek, mikroservis mimarisinin avantajlarını maksimize edebilir ve operasyonel verimliliği artırabilirsiniz.

6. Kullanılan Teknolojiler

Container teknolojileri (Docker , Kubernetes) , RESTful API tasarımı , Message queue sistemleri (Kafka , RabbitMQ) , Servis mesh(Örn. Istio)

-Container Teknolojileri (Docker, Kubernetes):

Docker: MikroServislerin hafif ve taşınabilir bir şekilde paketlenmesini ve dağıtılmasını sağlar.

Kubernetes: Docker konteynerlerini yönetmek, otomatikleştirmek ve ölçeklendirmek için kullanılan bir açık kaynaklı orkestrasyon sistemidir.

-RESTful API Tasarımı:

MikroServis mimarisinde genellikle RESTful API'lar kullanılır. Bu, HTTP protokolü üzerinden kaynaklara erişim sağlamak için standart bir yaklaşımdır.

- Message Queue Sistemleri (Kafka, RabbitMQ):

Kafka: Dağıtılmış akış işleme platformudur ve verilerin yüksek hızda işlenmesi ve dağıtılmasını sağlar.

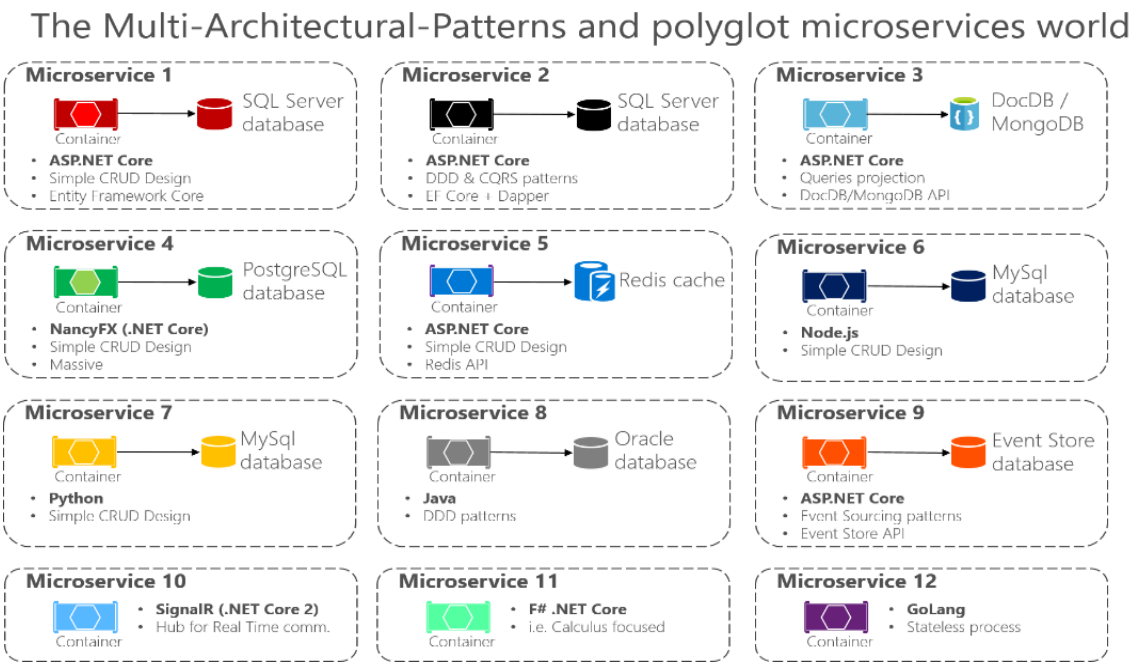
RabbitMQ: Birçok farklı protokol aracılığıyla mesajlaşmayı destekleyen bir mesaj yazılımıdır ve mikroServisler arasında iletişimi kolaylaştırır.

- Servis Mesh (Örn. Istio):

Istio: Servisler arası iletişimi güvenli, güvenilir ve hızlı hale getirmek için kullanılan bir servis mesh çözümüdür. MikroServislerin iletişimini yönetir, güvenlik politikalarını uygular ve trafik yönlendirmesi yapar.
Bu teknolojiler, mikroServis mimarisinin temel bileşenleri olarak kabul edilir ve genellikle birlikte kullanılırlar. Her biri, sistemdeki farklı ihtiyaçları karşılamak için özelleştirilebilir ve entegre edilebilir.

7. Mikroservis Uygulama Örneği

Basit bir e-ticaret uygulaması üzerinden örnek , servislerin birbirleriyle nasıl iletişim kurduğu , güncelleme ve yayınlama süreçleri



Basit E-Ticaret Mikroservis Uygulaması Tasarımı

Servislerin Tanımı

1. **Kullanıcı Yönetimi Servisi**
    - **İşlevselliği:** Kullanıcı kaydı, giriş, profil yönetimi gibi işlemleri sağlar.
    - **Teknolojiler:** Node.js ve Express kullanarak RESTful API sunabilir.
2. **Ürün Kataloğu Servisi**
    - **İşlevselliği:** Ürünleri listeler, detaylarını gösterir, arama yapılmasını sağlar.
    - **Teknolojiler:** Python ve Flask ile RESTful API sunabilir, veritabanı olarak SQLite veya MongoDB kullanabilir.
3. **Sipariş Yönetimi Servisi**
    - **İşlevselliği:** Sipariş oluşturma, iptal etme, durumunu güncelleme gibi sipariş yönetimi işlemlerini sağlar.
    - **Teknolojiler:** Java ve Spring Boot kullanarak RESTful API sunabilir, veritabanı olarak PostgreSQL veya MySQL kullanabilir.

Servisler Arası İletişim

Mikroservisler genellikle birbirleriyle HTTP üzerinden RESTful API'ler veya RPC (Remote Procedure Call) yöntemleriyle iletişim kurarlar. Örneğin:

- **Kullanıcı Yönetimi Servisi** kullanıcının kaydını aldığında veya güncellediğinde, **Sipariş Yönetimi Servisi**'ne bu bilgiyi iletebilir. Bu iletişim genellikle senkron veya asenkron olabilir. Senkron iletişimde doğrudan API çağrıları yapılabilirken, asenkron iletişimde bir mesaj kuyruğu (örneğin Kafka veya RabbitMQ) kullanılabilir.
- **Ürün Kataloğu Servisi** ürün bilgileri güncellendiğinde veya yeni bir ürün eklendiğinde, **Sipariş Yönetimi Servisi**'ne veya **Kullanıcı Yönetimi Servisi**'ne bu güncellemeleri yayınlayabilir. Bu, mikroservisler arasında bir olay yayınlama/mesajlaşma mekanizması kullanılarak gerçekleştirilebilir.

Güncelleme ve Yayınlama Süreçleri

- **Güncelleme:** Her bir servisin güncellemesi kendi süreçlerine göre yönetilir. Örneğin, bir servisin yeni bir özellik eklemesi veya bir hata düzeltmesi yapması durumunda, ilgili kod değişiklikleri yapılır ve versiyon kontrol sistemi (Git gibi) kullanılarak kaydedilir.
- **Yayınlama:** Güncelleme yapılan servis, konteyner teknolojileriyle (Docker, Kubernetes gibi) paketlenir ve bir konteyner imajı olarak yayınlanır. Bu imajlar, bir konteyner orkestrasyon aracı (örneğin Kubernetes) tarafından yönetilir ve dağıtılır. Yeni bir versiyon yayınlamak için genellikle sürekli entegrasyon/dağıtım (CI/CD) araçları kullanılır.

Örnek Senaryo

- **Senaryo:** Bir kullanıcı bir ürünü sepete eklediğinde:
    1. Kullanıcı, **Ürün Kataloğu Servisi**'ne ürün bilgisi talep eder.
    2. **Ürün Kataloğu Servisi**, ürün bilgisini döner.
    3. Kullanıcı, sepetine ürünü ekler ve siparişi tamamlamak istediğinde:
    4. **Sipariş Yönetimi Servisi**, yeni bir sipariş oluşturur ve durumu günceller.
    5. **Kullanıcı Yönetimi Servisi**, kullanıcının sipariş geçmişini günceller veya kredi kartı bilgilerini yönetir.

Bu örnek, basit bir e-ticaret uygulamasında mikroservislerin nasıl çalıştığını ve birbirleriyle nasıl etkileşimde bulunduklarını göstermektedir. Mikroservis mimarisi, her servisin bağımsız olarak geliştirilmesine ve ölçeklenmesine olanak tanırken, sistem genelindeki esnekliği ve hata toleransını artırır.

8. Sonuçlar ve Gelecek

Mikroservislerin geleceği , Daha fazla adaptasyon ve benimsenmesi beklenen alanlar , Geliştirme süreçlerine etkisi ve işletme maliyetleri üzerindeki etkiler

Daha Fazla Adaptasyon ve Benimsenme:

- Mikroservis mimarisi, ölçeklenebilirlik, esneklik ve hata toleransı gibi avantajlarıyla yazılım geliştiriciler ve işletmeler tarafından daha fazla benimsenmeye devam edecektir.
- Özellikle büyük ve karmaşık sistemlerin geliştirilmesi ve yönetilmesinde, mikroservisler modüler yapısıyla tercih edilmektedir.

Bulut ve Konteynerleşme ile Entegrasyon:

- Bulut bilişim ve konteynerleşme teknolojilerinin yaygınlaşması, mikroservislerin dağıtımı ve yönetimi için daha iyi altyapı sağlamaktadır.
- Kubernetes gibi konteyner orkestrasyon araçları, mikroservislerin kolayca yönetilebilmesini ve ölçeklenebilmesini sağlamaktadır.

Daha İleri Otomasyon ve DevOps Uygulamaları:

- CI/CD (Continuous Integration/Continuous Deployment) süreçlerinin daha yaygın olarak kullanılması, mikroservislerin hızlı bir şekilde güncellenmesini ve dağıtılmasını sağlamaktadır.
- Otomatik testler, güvenlik taramaları ve performans optimizasyonları gibi DevOps uygulamaları, mikroservis mimarisiyle birlikte entegre edilmektedir.

-Geliştirme Süreçlerine Etkisi

- **Modüler Geliştirme ve Bağımsızlıklar:** Her bir mikroservis, kendi işlevselliğini yönetir ve geliştirilir. Bu, geliştirme süreçlerinin daha modüler ve hızlı olmasını sağlar.
- **Daha Hızlı Dağıtım ve Güncelleme:** Servislerin bağımsız olarak dağıtılabilmesi ve güncellenebilmesi, hataların izole edilmesini ve hızlı geri dönüş sağlanmasını mümkün kılar.
- **Çevik Geliştirme ve İyileştirme:** Mikroservisler, yenilikçi özelliklerin hızla entegre edilmesine olanak tanır ve sürekli iyileştirme döngüsünü güçlendirir.

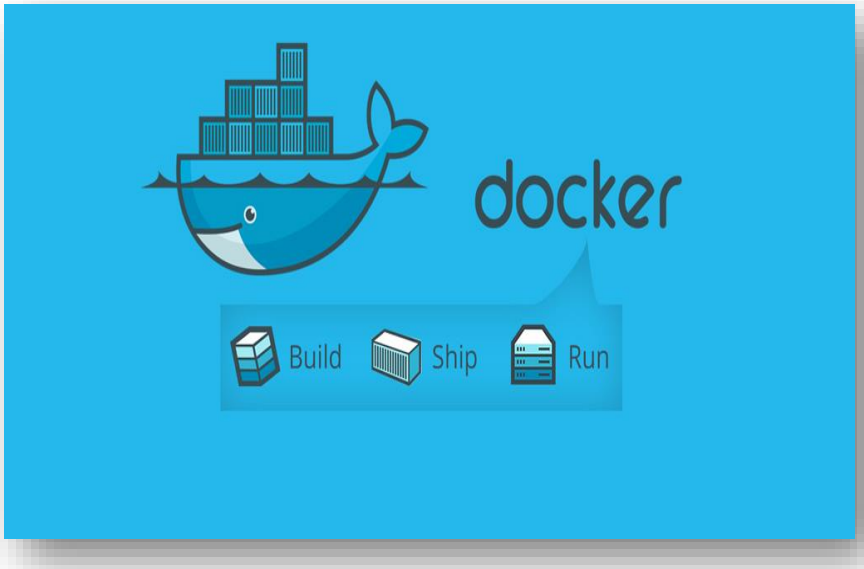
-İşletme Maliyetleri Üzerindeki Etkiler

- **Operasyonel Karmaşıklık:** Birden fazla mikroservisin yönetilmesi, operasyonel karmaşıklığı artırabilir. Ancak bu, doğru otomasyon ve yönetim araçları kullanılarak minimize edilebilir.
- **Yüksek Ölçeklenebilirlik ve Verimlilik:** Mikroservisler, gereksiz kaynak kullanımını önler ve sadece belirli bir servis için kaynakları ayırır. Bu, maliyetleri optimize eder ve ölçeklenebilirliği artırır.
- **Yatırım Getirisi:** Uzun vadede, mikroservis mimarisi genellikle işletmeler için daha iyi yatırım getirisi sağlar, çünkü esneklik, hızlı yenilik ve müşteriye odaklı çözümler sunar.

Sonuç olarak, mikroservislerin geleceği teknolojiye ve iş ihtiyaçlarına hızlı adapte olabilen, esnek ve ölçeklenebilir çözümler sunan bir mimari olarak görünmektedir. Ancak bu geçişin doğru planlama, yönetim ve yatırım gerektirdiği unutulmamalıdır.

DOCKER

1. Docker Nedir ? Temel Amacı , Konteynerizasyon kavramı ve avantajları



- Docker, yazılım uygulamalarını geliştirmek, dağıtmak ve çalıştırmak için kullanılan bir platformdur. Docker, uygulamaları konteyner adı verilen hafif, taşınabilir ve kapsayıcı birimlerde paketlemeye olanak tanır. Her bir Docker konteyneri, uygulamanın kodunu, bağımlılıklarını, kütüphanelerini ve yapılandırmasını içerir. Bu konteynerler, işletim sisteminden bağımsız olarak çalışabilir ve farklı ortamlarda kolayca taşınabilir ve dağıtılabilirler. Docker, konteyner teknolojisinin yaygınlaşmasını sağlamış ve yazılım geliştirme süreçlerini önemli ölçüde hızlandırmıştır. Konteynerler, uygulamaların geliştirilmesi, test edilmesi, dağıtılması ve ölçeklendirilmesi süreçlerini daha verimli ve güvenli hale getirir. Ayrıca, Docker'in kullanımı sayesinde yazılım uygulamalarının çalışma ortamları daha tutarlı hale gelir ve geliştirme ile operasyon ekipleri arasında daha iyi bir işbirliği sağlanabilir.

- Konteynerizasyon, bir yazılım uygulamasının çalışma ortamını izole etmek ve taşınabilir hale getirmek için kullanılan bir teknoloji ve yöntemdir. Bu teknoloji sayesinde uygulamalar, kapsayıcı birimlerde (konteynerlerde) paketlenir ve bu konteynerler, işletim sisteminden bağımsız olarak çalışabilirler. İşte konteynerizasyonun avantajları:

1. **Taşınabilirlik**: Konteynerler, uygulamaların kodunu, bağımlılıklarını ve yapılandırmalarını bir araya getirir. Bu sayede konteynerler, geliştirme ortamından üretim ortamına veya farklı bulut platformlarına kolayca taşınabilirler. Konteynerler, aynı anda farklı ortamlarda (örneğin, geliştirme, test ve üretim) tutarlı bir şekilde çalışabilir.

2. **Hafiflik ve Hız**: Konteynerler, sanal makinelerden daha hafif yapıdadır. Bir konteyner, kendi işletim sistemi çekirdeğine ihtiyaç duymadan, paylaşılan bir işletim sistemi üzerinde çalışabilir. Bu da konteynerlerin çok daha hızlı başlatılmasını ve durdurulmasını sağlar.

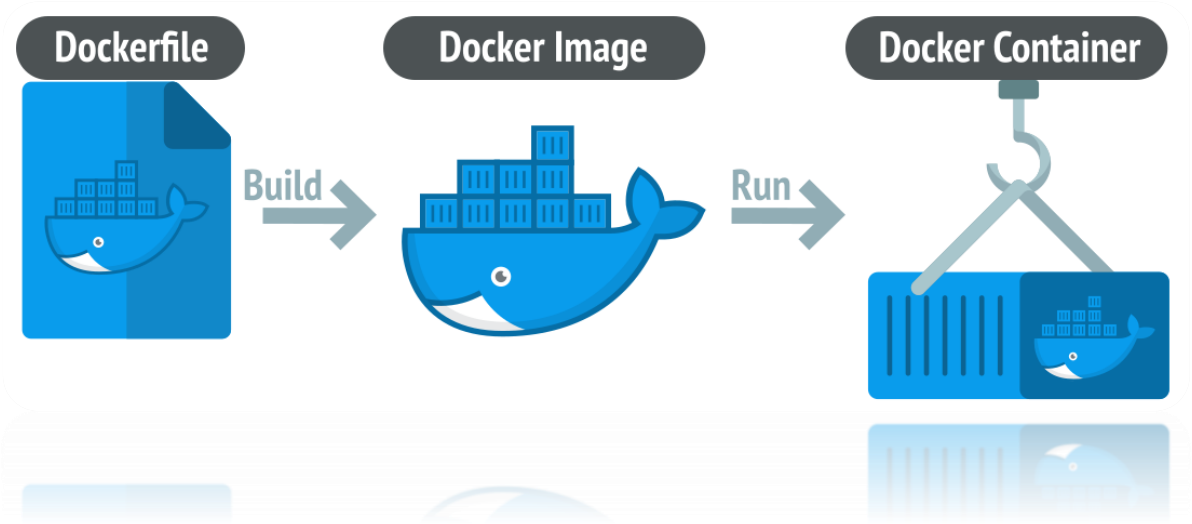
3. **İzolasyon**: Konteynerler, uygulamaları ve bunların bağımlılıklarını işletim sistemi seviyesinde izole eder. Bu sayede bir konteynerde yaşanan bir sorun, diğer konteynerleri etkilemez. Bu izolasyon, güvenlik açısından da önemli bir avantaj sağlar.

4. **Daha Kolay Yönetim ve Dağıtım**: Konteyner teknolojisi, uygulama ve altyapı bileşenlerinin bir arada yönetilmesini kolaylaştırır. Konteyner tabanlı uygulamalar, otomatik dağıtım ve ölçeklendirme işlemlerini destekler, bu da operasyonel verimliliği artırır.

5. **Çeviklik ve Esneklik**: Konteynerler, mikro-servis mimarileri için idealdir. Her bir servis veya bileşen, kendi konteynerinde çalışabilir ve bağımsız olarak geliştirilebilir, dağıtılabilir ve ölçeklendirilebilir. Bu da uygulama geliştirme süreçlerini daha çevik hale getirir.

6. **Kaynak Verimliliği**: Konteynerler, fiziksel ve sanal makinelerdeki kaynakların daha verimli kullanılmasını sağlar. Her bir konteyner, sadece ihtiyacı olan kaynakları alır ve paylaşılabilir, bu da sunucu kullanımını optimize eder.

Konteynerizasyon, modern yazılım geliştirme ve dağıtım süreçlerinde yaygın olarak kullanılmakta ve bu avantajlarıyla birlikte uygulamaların daha güvenli, hızlı ve etkili bir şekilde yönetilmesini sağlamaktadır.



2. Docker Temelleri

Docker Mimarisi: Docker Engine , Docker Daemon , Docker CLI – Docker Image ve Docker Container kavramları.

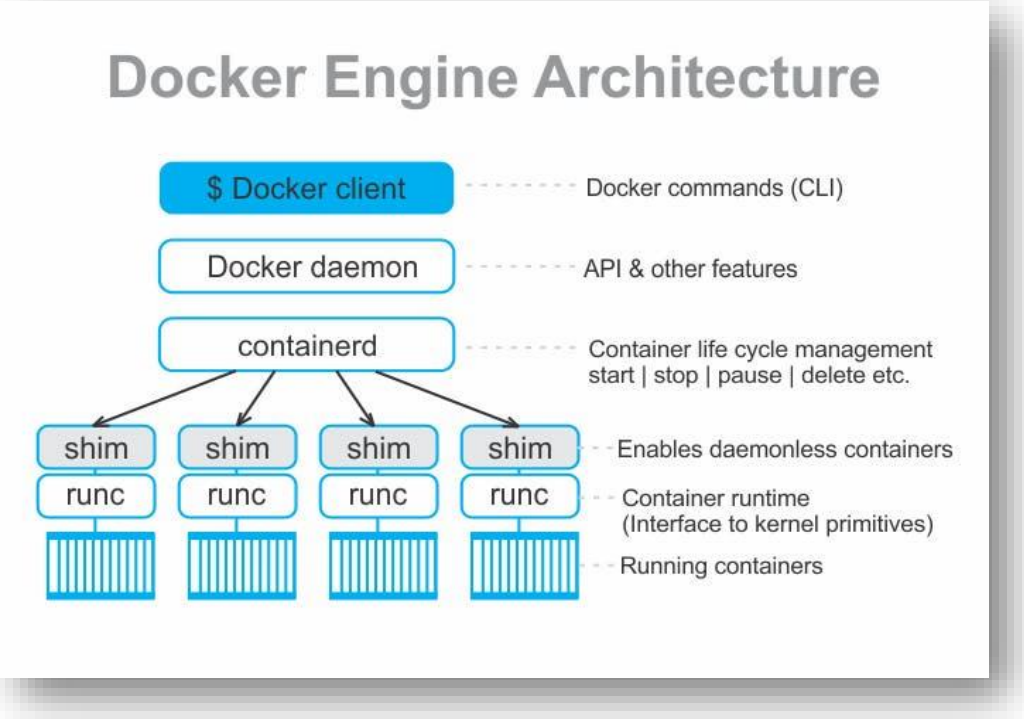
- Docker Mimarisi

Docker'ın mimarisi üç ana bileşenden oluşur:

1. **Docker Engine**: Docker'ın çekirdek bileşenidir ve Docker'ın çalışma zamanı ortamını sağlar. Docker Engine, Docker konteynerlerini yönetmek için gerekli olan API'leri sunar.

2. **Docker Daemon**: Arka planda çalışan bir süreçtir (arka plan servisi), Docker Engine'in temel bileşenidir ve Docker API isteklerini dinler. Docker Daemon, Docker konteynerlerinin oluşturulması, çalıştırılması, ağ yönetimi gibi işlemleri gerçekleştirir.

3. **Docker CLI (Command Line Interface)**: Komut satırı arayüzüdür ve Docker kullanıcıları tarafından Docker Engine ile iletişim kurmak için kullanılır. Kullanıcılar CLI aracılığıyla Docker komutlarını verir ve Docker Daemon üzerinde işlemler yaparlar.



- Docker Image ve Docker Container Kavramları

- **Docker Image**: Docker konteynerlerinin temelini oluşturan salt okunur dosyalardır. Bir Docker imajı, bir uygulamanın çalıştırılması için gerekli olan her şeyi içeren bir pakettir. Örneğin, bir işletim sistemi, yazılım uygulamaları, ve bağımlılıkları içerebilir. Docker imajları, Dockerfile adı verilen dosyalar kullanılarak oluşturulur.

- **Docker Container**: Bir Docker imajının çalışan bir örneğidir. Bir konteyner, bir işletim sistemi örneği ve bu işletim sistemi üzerinde çalışan uygulamaları içerir. Konteynerler, Docker Daemon tarafından yönetilir ve Docker imajlarından oluşturulur.

Docker konteynerleri, hafif ve taşınabilir olmaları sayesinde geliştirme, test etme ve dağıtma süreçlerini hızlandırır ve yazılım uygulamalarının çevreler arası geçişini kolaylaştırır.

3. Docker Container Yönetimi

Dockerfile: Image'lerin nasıl oluşturulduğu – Docker Registry: Image'lerin depolanması ve paylaşılması (Docker Hub , private registry)

- Dockerfile

Dockerfile, Docker imajlarının nasıl oluşturulacağını tanımlayan metin tabanlı bir dosyadır. İmaj oluşturma işlemi adımlar halinde bu dosyada belirtilir ve Docker Engine tarafından okunarak işlenir. İşte bir Dockerfile içerebilecek temel bileşenler:

1. **Base Image**: Docker imajının temelini oluşturan başka bir imajı belirler. Örneğin, bir Linux dağıtımı veya başka bir yazılım platformu.

2. **Komutlar**: İmaj oluşturma işlemleri için çalıştırılacak komutlar. Örneğin, paket yükleme komutları, dosya kopyalama, ortam değişkenleri ayarlama gibi işlemler burada yapılır.

3. **Çalıştırılacak Komut**: Konteyner başlatıldığında otomatik olarak çalışacak komut veya komutlar.

Bir Dockerfile örneği:

```
# Base image
FROM ubuntu:20.04

# Update package lists
RUN apt-get update

# Install necessary packages
RUN apt-get install -y nginx

# Copy configuration file
COPY nginx.conf /etc/nginx/nginx.conf

# Expose port 80
EXPOSE 80

# Define the command to start nginx
CMD ["nginx", "-g", "daemon off;"]
```

Bu Dockerfile, Ubuntu 20.04 tabanlı bir imaj oluşturur, Nginx web sunucusunu yükler, bir yapılandırma dosyasını kopyalar, port 80'i açar ve nginx'i başlatır.

Docker Registry

Docker Registry, Docker imajlarının depolandığı ve paylaşıldığı bir yerdir. En yaygın kullanılan Docker Registry hizmeti Docker Hub'dır, ancak kendi özel Docker Registry'nizi de kurabilirsiniz.

- **Docker Hub**: Docker topluluğu tarafından sağlanan ve geniş bir halka açık Docker imajları koleksiyonunu barındıran bir bulut tabanlı Docker Registry hizmetidir. Burada birçok popüler açık kaynak proje ve yazılım imajları bulunabilir.
- **Private Registry**: Güvenlik veya özel uygulama gereksinimleri nedeniyle kendi kurum içi veya bulut tabanlı özel Docker Registry'lerini kullanabilirsiniz. Bu, Docker imajlarınızı kuruluş içinde veya belirli kullanıcılar arasında paylaşmanıza ve depolamanıza olanak tanır.

Docker imajlarını oluşturduktan sonra, bu imajları Docker Registry'ye (örneğin Docker Hub veya özel Registry) yükleyebilirsiniz. İmajlar, Docker CLI veya Docker API aracılığıyla Registry'den çekilebilir veya Registry'ye yüklenir. Bu sayede ekip üyeleri veya sistemler arası imaj paylaşımı ve dağıtımı kolaylaşır.

Özel Docker Registry Kurulumu

1. **Docker Registry Kurulumu**: Docker Registry, Docker resmi imajı olarak Docker Hub'da bulunur. Kurulum için aşağıdaki komutu kullanabilirsiniz:

```bash
docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

Bu komut, Docker Hub'dan registry:2 imajını indirir ve yerel bir Docker konteyneri olarak çalıştırır. 5000 portu üzerinden erişilebilir. --restart=always seçeneği, Docker servisinin yeniden başlatıldığında otomatik olarak çalışmasını sağlar.

2. **Docker Registry Konfigürasyonu**: Varsayılan olarak, Docker Registry herkese açık olarak başlatılır. Güvenlik veya erişim kontrolleri sağlamak için, SSL/TLS sertifikaları kullanabilir veya kimlik doğrulama ekleyebilirsiniz. Bu adımlar, config.yml dosyası üzerinden yapılandırılabilir.

3. **Docker Imajlarınızı Yükleyin ve Kullanın**: Docker CLI aracılığıyla imajları özel Registry'ye yükleyebilirsiniz. Örneğin:

```bash
# Docker imajını özel Registry'ye yükle
docker tag <existing-image> localhost:5000/myimage
docker push localhost:5000/myimage
```

‘<existing-image>’ mevcut Docker imajınızın adıdır. ‘localhost:5000’ adresi, yerel Docker Registry'nizin adresidir.

Imajları Çekme: Docker Registry'den imajları çekmek için:

```bash
docker pull localhost:5000/myimage
```

Bu komut, ‘localhost:5000’ adresinden ‘myimage’ isimli Docker imajını çeker.

Güvenlik Dikkatleri

Özel Docker Registry kullanırken güvenlik önlemlerini ihmal etmemek önemlidir. SSL/TLS sertifikaları kullanarak iletişimi şifrelemek ve kimlik doğrulama (auth) özelliğini etkinleştirmek önerilir. Böylece, imajların sadece yetkili kullanıcılar tarafından erişilebilir olmasını sağlayabilirsiniz.

Özetle, Docker Registry kullanarak kendi özel Docker imajlarınızı yönetmek ve dağıtmak, güvenliği sağlamak ve geliştirme süreçlerini iyileştirmek için etkili bir yöntemdir.

4. Docker Networking

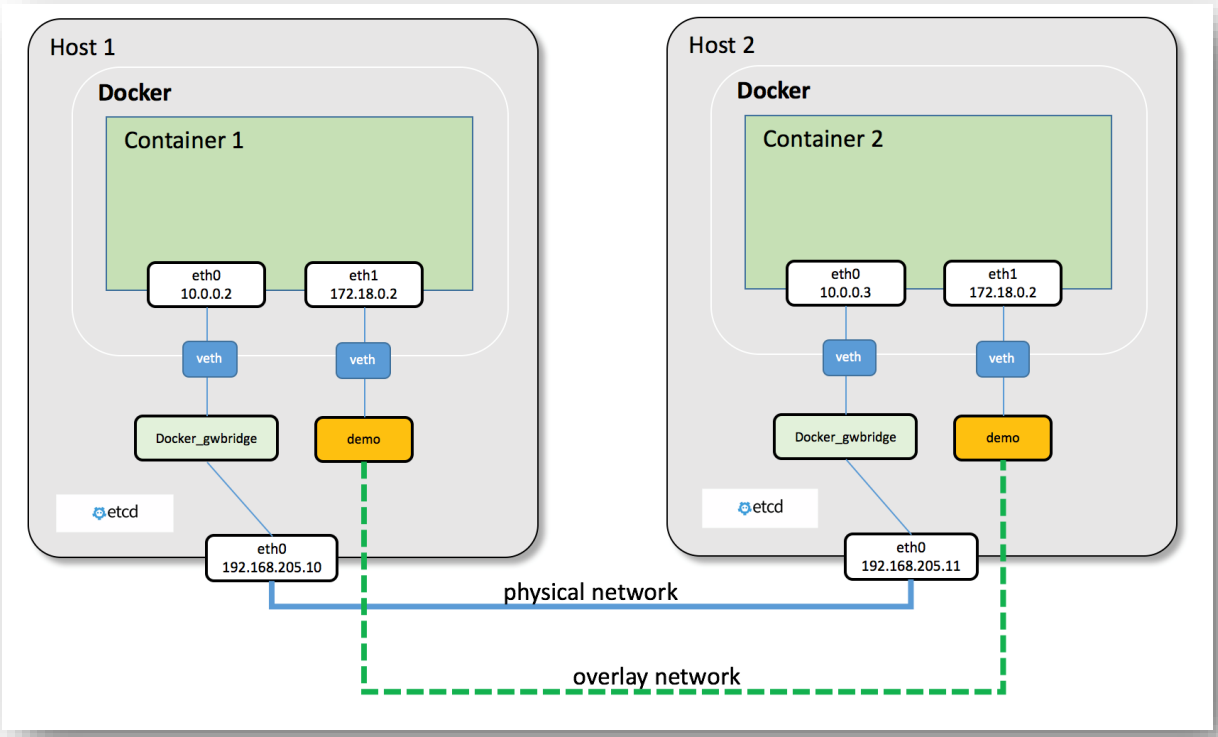
Konteynerler arası iletişim ve ağ konfigürasyonu - Bridge network , host network , overlay network gibi Docker ağ modelleri.

1. **Bridge Network (Köprü Ağı)**:
    - Docker varsayılan olarak köprü ağı (bridge network) adı verilen bir sanal ağ oluşturur.
    - Her Docker daemon'ı, köprü ağı üzerindeki konteynerler için bir köprü cihazı oluşturur.
    - Köprü ağı, aynı Docker ana makinesinde çalışan konteynerler arasında iletişimi sağlar.
    - Konteynerler, varsayılan olarak bu köprü ağına bağlanır ve Docker daemon'ı, bu ağda IP adresleri atar.

2. **Host Network (Ana Makine Ağı)**:
    - Konteynerlerin Docker ana makinesinin ağına doğrudan bağlandığı bir ağ modudur.
    - Bu modda, konteynerler Docker host'unun ağına dahil edilir, dolayısıyla host'un IP adresleri ve ağ ayarları doğrudan konteynerlere uygulanır.
    - Performans açısından avantaj sağlar, çünkü ağ trafiği Docker ağından geçmeden doğrudan ana makine ağı üzerinden iletilir.

3. **Overlay Network (Örtü Ağı)**:
    - Çoklu Docker daemon'ı (hostlar arası) üzerinde çalışan konteynerler arasında iletişim sağlamak için kullanılır.
    - Overlay ağı, Docker Swarm gibi bir container orchestration (konteyner yönetimi) aracıyla kullanılır.
    - Bu ağ modeli, farklı fiziksel veya sanal ağlarda çalışan Docker konteynerlerinin iletişimini sağlamak için kullanılır.
    - Overlay ağlar, birden fazla Docker host arasında güvenli iletişim ve ağ segmentasyonu sağlamak için şifreleme gibi ek özellikler sunabilir.

Bu ağ modelleri, Docker konteynerlerinin belirli ihtiyaçlarına göre seçilir ve konfigüre edilir. Köprü ağı genellikle standart konteynerler için tercih edilirken, performans gereksinimleri yüksek olan uygulamalar için host ağı veya büyük ölçekli dağıtımlar için overlay ağı tercih edilebilir. Her bir ağ modelinin avantajları ve kullanım senaryoları farklı olduğundan, doğru modelin seçilmesi uygulamanın ihtiyaçlarına bağlıdır.



5. Docker Depolama Yönetimi

Volume ve Volume Tipleri (bind mounts , named volumes) – Data persistence ve containerler arası veri paylaşımı.

Docker'da depolama yönetimi, özellikle veri kalıcılığı (data persistence) ve containerler arası veri paylaşımı konularında önemli bir rol oynar. Docker'da bu amaçla kullanılan temel kavramlar Volumes (hacimler) ve Volume Tipleri olarak adlandırılır. İki yaygın volume tipi bind mounts ve named volumes şeklindedir.

1. Bind Mounts (Bağlama Bağlantı Noktaları)

Tanım: Bir host dosya veya dizinini doğrudan bir container dosya sistemi yoluna bağlamak için kullanılan bir mekanizmadır.

Kullanım:

- Host dosya sisteminden container dosya sistemine doğrudan erişim sağlar.
- Docker CLI veya Docker Compose ile tanımlanabilir.

Özellikler:

- Esneklik: Host dosyalarını ve dizinlerini container'a direkt olarak bağlar.
- Performans: Host dosya sistemi performansı, container performansını etkileyebilir.
- Kolaylık: Geliştirme aşamasında sık kullanılır, fakat production ortamlarında dikkatli kullanılmalıdır.

```bash
docker run -v /host/path:/container/path ...
```

2. Named Volumes (İsimlendirilmiş Hacimler)

Tanım: Docker tarafından yönetilen ve özel bir veritabanında saklanan depolama alanlarıdır.

Kullanım:

- Docker tarafından oluşturulur, yönetilir ve kullanılır.
- Volume'ler, containerler arasında veri paylaşımı ve veri kalıcılığı sağlamak için idealdir.

Özellikler:

- Yönetim: Docker CLI veya Docker Compose aracılığıyla yönetilir.
- Güvenlik: Container'lar arasında izole edilmiş veri saklama sağlar.
- Performans: Docker, verileri optimize ederek çeşitli depolama sürücülerini destekler.

```bash
docker volume create my_volume
docker run -v my_volume:/container/path ...
```

Veri Kalıcılığı ve Containerler Arası Veri Paylaşımı

- **Veri Kalıcılığı:** Docker container'ları, örneğin bir veri tabanı veya uygulama sunucusu, yeniden başlatıldığında veya yeniden oluşturulduğunda, verileri kaybetmemesi için volume'ler üzerinde çalışır.

- **Containerler Arası Veri Paylaşımı:** Named volumes, farklı container'lar arasında veri paylaşımı sağlar. Örneğin, bir veri tabanı container'ı verileri bir named volume üzerinde saklar ve diğer container'lar bu volume'ü kullanarak verilere erişir.

Docker'da volume'ler, veri kalıcılığı ve containerler arası iletişimi sağlamak için kritik bir rol oynar. Hangi volume tipinin kullanılacağı, proje gereksinimlerine ve kullanım senaryolarına bağlı olarak belirlenmelidir.

6. Docker Compose

Birden fazla container'in birlikte çalıştırılması ve yönetimi için YAML tabanlı tanımlama – Servis tanımları , network ve volume konfigürasyonları.

Docker Compose, birden fazla Docker container'ının birlikte çalıştırılması ve yönetilmesi için kullanılan YAML tabanlı bir araçtır. Docker Compose kullanarak, bir veya birden fazla servisin (container'ın) tanımlarını ve bu servisler arasındaki ağ yapılandırmalarını, disk kullanımı (volume) konfigürasyonlarını kolayca yönetebilirsiniz.

Docker Compose'un YAML dosyasında genellikle şunlar tanımlanır:

1. **Servis Tanımları**: Her bir container (servis) için hangi Docker imajının kullanılacağı, hangi portların açılacağı, hangi ortam değişkenlerinin ayarlanacağı gibi detaylar burada belirtilir.

2. **Networks (Ağlar)**: Docker Compose ile oluşturduğunuz container'lar arasındaki ağ konfigürasyonlarını yönetebilirsiniz. Örneğin, container'lar arasında iletişim kurulabilmesi için aynı ağı kullanmalarını sağlayabilirsiniz.
3. **Volumes (Diskler)**: Veri kalıcılığını sağlamak için kullanılan disk (volume) konfigürasyonlarını tanımlar. Örneğin, bir veri tabanı servisinin verilerini saklamak için bir volume tanımlayabilirsiniz.

Docker Compose kullanarak bu tanımlamaları yapar ve ardından docker-compose up komutunu kullanarak tüm servisleri ayağa kaldırabilirsiniz. Bu komut, Docker Compose dosyasındaki tanımlamalara göre tüm container'ları oluşturur ve başlatır.

Örneğin, aşağıdaki gibi bir Docker Compose YAML dosyası, bir web uygulaması ve bir veri tabanı servisini tanımlar:

```yaml
version: '3'
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
  db:
    image: mysql:latest
    environment:
      MYSQL_ROOT_PASSWORD: example
    volumes:
      - db_data:/var/lib/mysql
volumes:
  db_data:
```

Bu YAML dosyasında web ve db adında iki servis tanımlanmıştır. web servisi nginx imajını kullanarak 80 numaralı portu host'a bağlar, db servisi ise mysql imajını kullanarak bir MySQL veri tabanı oluşturur. Ayrıca, db servisinin veri tabanı verilerini saklamak için bir volume tanımlanmıştır. Docker Compose, bu tanımlamalara göre hem web hem de db servislerini ayağa kaldırmak için kullanılabilir.

7. Docker Güvenlik

Konteyner izolasyonu ve güvenlik önlemleri – Docker Security Best Practices.

Docker konteynerlerinin güvenliği, doğru yapılandırma ve güvenlik önlemlerinin uygulanmasıyla sağlanabilir. İşte Docker güvenliği için en iyi uygulamalar:

1. **İzolasyon ve Konteyner Ayarları:**
   - Konteynerler arasında güçlü izolasyon sağlamak için Docker'ın varsayılan ayarlarını kullanın.
   - Çekirdek adımını sınırlayın (--kernel-memory, --memory, --cpu-shares, vb.) ve gereksiz sistem çağrılarını engelleyin (--no-new-privileges).

2. **Güvenli Geliştirme ve Dağıtım Uygulamaları:**
   - Güvenli Docker imajları oluşturmak için temiz ve güvenilir temel imajlar kullanın.
   - Gereksiz bileşenleri kaldırın ve sadece gerekli olanları dahil edin.
   - Güvenlik açıkları için düzenli olarak imajları tarayın ve güncelleyin.

3. **Ağ Güvenliği:**
   - Ağ ayarlarını doğru yapılandırın. Konteynerler arası iletişimi gereksiz yere açmayın.
   - Gerekirse, konteynerlere ağ izolasyonu uygulayın ve ağ politikaları oluşturun (--network, --link, vb.).

4. **Erişim Yönetimi ve İzleme:**
   - Güçlü parolalar ve kimlik doğrulama yöntemleri kullanarak konteynerlere erişimi sınırlayın.
   - Docker Daemon'a erişimi sınırlayın ve yetkilendirme gereksinimlerini uygulayın.
   - Konteyner faaliyetlerini ve günlükleri izleyin (docker logs, Docker Daemon günlükleri, vb.).

5. **Güvenlik Eklentileri ve Araçlar:**
   - Docker güvenliğini artırmak için güvenlik eklentileri ve araçları kullanın.
   - Örneğin, Docker Bench for Security gibi araçlarla güvenlik denetimleri yapın ve önerilen yapılandırmalara uygunluğu kontrol edin.

6. **Güncel Kalın:**
   - Docker Daemon, konteyner imajları ve bağımlılıklarını düzenli olarak güncelleyin.
   - Güncellemeler, güvenlik açıklarını düzeltmek için önemlidir.

7. **Koruma ve Yedekleme Stratejileri:**
   - Konteyner verilerini düzenli olarak yedekleyin ve kritik bilgileri korumak için gerekli güvenlik önlemlerini alın.
   - Veri kaybını önlemek için uygun yedekleme stratejileri geliştirin.

Docker konteynerlerinin güvenliği, bu tür en iyi uygulamaların sistematik bir şekilde uygulanmasıyla artırılabilir. Her uygulama için özel gereksinimler olabilir, bu nedenle güvenlik stratejinizi iş yüklerinize ve ortamınıza özgü olarak ayarlamak önemlidir.

8. Docker İzleme ve Günlükleme

Konteynerlerin izlenmesi ve performans takibi – Docker log yönetimi.

Konteyner İzleme ve Performans Takibi

1. **Docker Stats Komutu**: Docker'in ‘stats’ komutu, çalışan konteynerlerin CPU kullanımı, bellek tüketimi, ağ aktivitesi gibi canlı performans verilerini gösterir.

```bash
docker stats [container_name or container_id]
```

cAdvisor: Google tarafından geliştirilen cAdvisor (Container Advisor), Docker konteynerlerinin performansıyla ilgili detaylı metrikler sağlar. Docker'ın dahili web arayüzünde veya Prometheus gibi bir monitörleme aracı üzerinden erişilebilir.

Docker Events: Docker daemon üzerinde meydana gelen olayları dinlemek için docker events komutu kullanılır. Bu komut, konteyner oluşturma, durdurma, başlatma gibi eylemleri takip etmenizi sağlar.

Docker API ve CLI ile Özel İzleme Araçları: Docker API veya CLI ile kendi izleme araçlarınızı oluşturabilirsiniz. Örneğin, Python veya Bash kullanarak belirli metrikleri toplayabilir ve analiz edebilirsiniz.

Docker Log Yönetimi

1. **Logları İzleme**: Docker konteynerlerinin loglarını izlemek için docker logs komutunu kullanabilirsiniz. -f (follow) seçeneği ile canlı olarak güncellemeleri izleyebilirsiniz.

2. **Log Rotasyonu**: Docker konteynerlerinin loglarının büyümesini önlemek için log rotasyonunu yapılandırabilirsiniz. Bu, log dosyalarının boyut veya yaşa göre nasıl yönetileceğini belirtir.

Log rotasyonu için Docker daemon ayarlarındaki log-driver ve log-opts parametreleri kullanılır.

3. **Loglarını Dışa Aktarma**: Konteyner loglarını Docker dışına çıkarmak ve merkezi bir log yönetim sistemine yönlendirmek için çeşitli yöntemler vardır. Bu, loglarınızı analiz etmek, depolamak ve erişmek için daha iyi bir yönetim sağlar.

- Syslog veya ELK (Elasticsearch, Logstash, Kibana) gibi araçlar kullanarak merkezi log toplama ve analiz yapabilirsiniz.

4. **Log Formatı ve Etiketleme**: Logların okunabilirliğini ve anlamlılığını artırmak için uygun formatlama ve etiketleme kullanılmalıdır. Örneğin, loglarda tarih, saat, konteyner adı gibi bilgilerin bulunması önemlidir.

9. Docker Orkestrasyon Araçları

Kubernetes , Docker Swarm gibi orkestrasyon sistemleri ile Docker kullanımı – Container orchestrators ile yönetim avantajları ve seçim kriterleri.

Docker, konteyner teknolojisinin popüler bir uygulayıcısıdır ve büyük ölçekli uygulamaları yönetmek için Docker Swarm ve Kubernetes gibi orkestrasyon araçları kullanılabilir. İşte bu iki orkestrasyon aracının Docker kullanımı üzerindeki avantajları ve seçim kriterleri:

Docker Swarm

Docker Swarm, Docker tarafından geliştirilen ve Docker konteynerlerini yönetmek için yerleşik bir orkestrasyon aracıdır. Temel avantajları şunlardır:

1. **Kolay Kurulum ve Yönetim**: Docker Swarm, Docker Engine ile entegre olduğu için kurulumu ve yönetimi Docker kullanıcıları için daha basittir. Docker komut satırı arayüzü üzerinden kolayca yönetilebilir.

2. **Düşük Karmaşıklık**: Küçük ve orta ölçekli uygulamalar için uygun olan basit bir yapıya sahiptir. Bu nedenle Docker kullanıcıları için hızlı bir şekilde benimsenebilir.

3. **Docker Hub Entegrasyonu**: Docker Swarm, Docker Hub entegrasyonuyla geliyor ve Docker imajlarını yönetmek ve paylaşmak için bu entegrasyon avantaj sağlıyor.

4. **Performans**: Docker Swarm, Swarm modunda çalışarak Docker konteynerlerinin performansını artırabilir ve yük dengeleme yapabilir.

Kubernetes

Kubernetes (k8s), Google tarafından geliştirilen ve açık kaynaklı bir konteyner orkestrasyon platformudur. Başlıca avantajları şunlardır:

1. **Ölçeklenebilirlik**: Kubernetes, çok büyük ölçekli uygulamaların yönetimi için tasarlanmıştır. Binlerce konteynerı kolayca yönetebilir ve otomatik olarak ölçeklendirebilir.

2. **Geniş Ekosistem**: Kubernetes, geniş bir ekosisteme sahiptir ve birçok cloud sağlayıcısı ve araçlarla entegre çalışabilir. Ayrıca birçok üçüncü taraf eklentisi ve araçları destekler.

3. **Yüksek Erişilebilirlik**: Kubernetes, uygulamaların yüksek erişilebilirlik gereksinimlerini karşılamak için hazır yapılandırmalar sunar ve otomatik olarak hizmetleri denetler.

4. **Hareketli İş Yükleri**: Kubernetes, mikro hizmetlerin yönetimini kolaylaştırır ve farklı ortamlar arasında (yerel, bulut, hibrit) uygulama taşımak ve çalıştırmak için esneklik sağlar.

Seçim Kriterleri

Hangi orkestrasyon aracının seçileceği, projenin gereksinimlerine ve organizasyonun teknik yeteneklerine bağlıdır. İşte bazı seçim kriterleri:

1. **Ölçek**: Büyük ölçekli bir uygulama mı yönetiyorsunuz yoksa küçük/orta ölçekli mi? Kubernetes büyük ölçekli uygulamalar için genellikle daha uygunken, Docker Swarm küçük/orta ölçekli uygulamalar için daha basit bir seçenek olabilir.

2. **Kurulum ve Yönetim**: Docker ekosistemi içinde mi kalma eğilimindesiniz yoksa daha geniş bir açık kaynak ekosistemi mi tercih ediyorsunuz?

3. **Eğitim ve Beceri**: Ekipteki yetenekler ve deneyim seviyesi hangi orkestrasyon aracının seçilmesinde önemli bir faktördür. Docker ile zaten deneyimli bir ekip Docker Swarm'a daha çabuk adapte olabilirken, Kubernetes'in daha derin bilgi gerektirebileceğini unutmamak önemlidir.

4. **Ekosistem ve Entegrasyon**: Hangi cloud sağlayıcılarıyla ve diğer araçlarla entegre olmanız gerekiyor? Kubernetes geniş bir entegrasyon sağlarken, Docker Swarm bu konuda daha sınırlı olabilir.

Sonuç olarak, Docker Swarm ve Kubernetes arasında seçim yaparken projenizin ölçeği, karmaşıklığı, ekosistem gereksinimleri ve ekibinizin deneyim seviyesi gibi faktörleri dikkate almalısınız. Her iki platform da farklı avantajlar sunar ve doğru seçim, organizasyonunuzun ihtiyaçlarına en iyi şekilde cevap verecek olanıdır.

10. Docker ve CI/CD Entegrasyonu

Docker’in sürekli entegrasyon ve dağıtım (CI/CD) süreçlerindeki rolü – Docker imajlarının otomatik olarak oluşturulması ve dağıtılması.

Docker, sürekli entegrasyon ve dağıtım (CI/CD) süreçlerinde önemli bir rol oynamaktadır çünkü yazılım geliştirme ve dağıtım süreçlerini daha hızlı, güvenilir ve tekrarlanabilir hale getirir. İşte Docker'in CI/CD süreçlerindeki rolü ve Docker imajlarının nasıl otomatik olarak oluşturulup dağıtıldığına dair genel bir açıklama:

Docker'in CI/CD Süreçlerindeki Rolü

1. **Ortam Standartlığı ve İzolasyonu:** Docker, yazılımı bir konteyner içinde paketleyerek, geliştirme, test ve üretim ortamları arasında tutarlılık sağlar. Bu, uygulamaların her ortamda aynı şekilde çalışmasını ve beklenen sonuçları vermesini sağlar.

2. **Hızlı Dağıtım:** Docker konteynerleri, hızlı bir şekilde oluşturulabilir, dağıtılabilir ve ölçeklenebilir. Bu, yeni özelliklerin veya güncellemelerin hızla kullanıcıya ulaştırılabilmesini sağlar.

3. **Otomasyon ve Tekrarlanabilirlik:** Docker imajları ve konteynerler, YAML veya JSON gibi dosya formatlarıyla tanımlanabilir ve bu tanımlamalar CI/CD araçları (örneğin Jenkins, GitLab CI/CD, CircleCI) tarafından otomatik olarak işlenebilir. Bu sayede dağıtım süreçleri tekrarlanabilir ve güvenilir hale gelir.

4. **Yazılım Dağıtımının Güvenliği:** Docker, imajlar arasında bir işlem sırası oluşturarak, güvenlik kontrolleri ve imza doğrulamaları gibi güvenlik önlemlerini otomatikleştirmeyi sağlar. Bu, güvenlik açıklarını tespit etmek ve olası saldırıları önlemek için önemli bir mekanizma sunar.

Docker İmajlarının Otomatik Oluşturulması ve Dağıtılması

1. **Otomatik İmaj Oluşturma:** Yazılım geliştirme sürecinde her commit veya belirli branch'lerde değişiklik yapıldığında, CI/CD araçları Docker imajlarının otomatik olarak oluşturulmasını tetikleyebilirler. Bu imajlar genellikle Dockerfile ve bağımlılıkların tanımlandığı dosyalar kullanılarak inşa edilir.

2. **Test ve Doğrulama:** Oluşturulan Docker imajları, test aşamasından geçirilir. Bu aşamada uygulamanın doğru çalıştığı ve beklenen sonuçları verdiği doğrulanır. Otomatik testler, birim testleri, entegrasyon testleri ve performans testleri gibi farklı test türleri kullanılabilir.

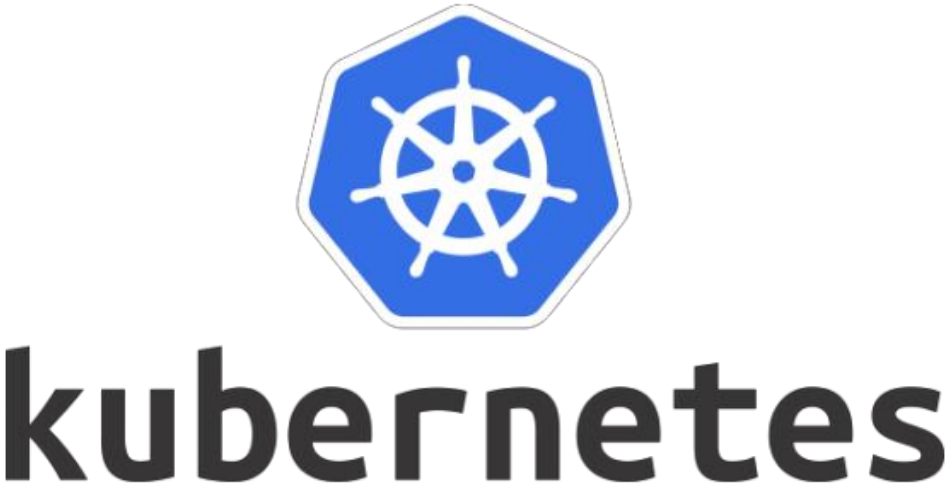
3. **Imaj Depolama ve Dağıtım:** Başarıyla geçen testlerden sonra, Docker imajları Docker Registry gibi bir imaj depolama hizmetine yüklenir. Docker Registry, imajların merkezi bir konumda saklanmasını ve yönetilmesini sağlar.

4. **Dağıtım Aşaması:** CI/CD sürecinin dağıtım aşamasında, imajlar hedef ortamlara (örneğin üretim sunucuları, test sunucuları) dağıtılır. Bu dağıtım işlemi, Kubernetes gibi bir konteyner yönetim platformu üzerinde yapılabileceği gibi, manuel olarak da olabilir.

5. **Sürekli Dağıtım:** CI/CD süreci bu şekilde yapılandırıldığında, her yeni commit veya değişiklik sonrası otomatik olarak yeni bir Docker imajı oluşturulabilir ve dağıtılabilir. Bu süreç, yazılım geliştirme sürecinin hızını artırır ve insan hatalarını azaltır.

Docker, bu süreçlerin otomatikleştirilmesi ve standartlaştırılmasını sağlayarak, yazılım geliştirme ekibinin verimliliğini artırır ve uygulamaların hızlı bir şekilde ve güvenli bir şekilde dağıtılmasını mümkün kılar.

KUBERNETES



1. Giriş ve Tanıtım

- Kubernetes nedir ? Temel Amacı

Kubernetes (k8s), konteynerlerinizi yönetmek ve uygulamalarınızı dağıtmak için kullanılan açık kaynaklı bir platformdur. Temel amacı, konteynerleştirilmiş uygulamaların otomatik olarak dağıtılması, ölçeklendirilmesi ve yönetilmesini sağlamaktır. Kubernetes, özellikle mikro hizmet mimarilerinde ve dağıtık sistemlerde kullanım için tasarlanmıştır.

Kubernetes'in sağladığı başlıca özellikler şunlardır:

1. **Otomatik dağıtım ve ölçekleme:** Uygulamalarınızı Kubernetes üzerinde tanımladığınızda, Kubernetes bu uygulamaları belirlediğiniz durumlar doğrultusunda otomatik olarak dağıtır ve ölçekler.

2. **Hizmet Keşfi ve Duyarlılık:** Servisler arasında iletişimi yönetmek için DNS tabanlı bir hizmet keşif mekanizması sunar. Bu sayede, servislerin birbirini bulması ve iletişim kurması kolaylaşır.

3. **Depolama Orkestrasyonu:** Farklı depolama çözümlerini yönetir ve konteynerlerin depolama kaynaklarına erişimini sağlar.

4. **Otomatik Rolback:** Uygulamalarınızda bir sorun oluştuğunda, Kubernetes otomatik olarak önceki sağlam duruma geri dönebilir.

5. **Kapsayıcı güvenlik:** Kubernetes, izolasyon ve güvenlik özellikleri ile konteynerler arasında güvenliği sağlar.

6. **Yük dengeleme:** Gelen istekleri otomatik olarak konteynerler arasında dağıtarak yükü dengeleyebilir.
7. **Bütünleşik sağlık kontrolü:** Uygulamalarınızın durumunu izler ve otomatik olarak yeniden başlatma veya otomatik olarak ölçeklendirme gibi eylemler yapabilir.

Kubernetes'in bu özellikleri, uygulamalarınızı daha güvenli, ölçeklenebilir ve yönetilebilir hale getirir. Bu nedenle, modern bulut uygulamalarını geliştirirken ve işletirken Kubernetes oldukça önemli bir araç haline gelmiştir.

- *Konteyner Teknolojileri ve Kubernetes Arasındaki İlişki*
Konteyner teknolojileri ve Kubernetes arasındaki ilişki oldukça yakındır, ancak farklı seviyelerde işlev görürler:

1. **Konteyner Teknolojileri (Docker, Containerd, etc.):**
    - Konteyner teknolojileri, uygulamaları hafif ve taşınabilir bir şekilde paketlemek için kullanılır. Örneğin, Docker konteynerleri, uygulama ve bağımlılıklarını tek bir paket içinde izole eder ve bu paketin herhangi bir sistemde çalışmasını sağlar.
    - Konteyner teknolojileri, işletim sistemi seviyesinde sanallaştırma kullanarak, uygulamalar arasında kaynak izolasyonu sağlar ve güvenli bir ortam sunar.

2. **Kubernetes:**
    - Kubernetes, konteyner teknolojilerinin yönetimini kolaylaştıran bir platformdur. Birden fazla konteyneri bir araya getirir ve bu konteynerlerin birlikte çalışmasını, otomatik olarak dağıtılmasını, ölçeklenmesini ve yönetilmesini sağlar.
    - Kubernetes, konteynerlerin çalıştırılması için fiziksel veya sanal makinelerde bir ortam sağlar. Konteynerlerin düzenlenmesi, durumlarının izlenmesi, otomatik olarak yeniden başlatılması gibi operasyonel görevleri de üstlenir.
    - Ayrıca Kubernetes, hizmet keşfi, yük dengeleme, depolama yönetimi gibi ek özellikler sunarak uygulama geliştiricilerine daha fazla esneklik ve operasyonel kolaylık sağlar.

Kısacası, konteyner teknolojileri uygulamaları izole ederek hafif ve taşınabilir hale getirirken, Kubernetes bu konteynerleri yönetmek, orkestre etmek ve büyük ölçekte dağıtmak için bir platform sağlar. Kubernetes, birçok konteynerin bir arada çalışmasını sağlamak için geliştirilmiş karmaşık sistemler üzerinde çalışabilirken, konteyner teknolojileri daha çok bir konteynerin nasıl oluşturulduğu ve çalıştırıldığına odaklanır.

2. Kubernetes Temelleri

- *Kubernetes Mimarisi: Master ve Worker node'lar.*

Kubernetes'in temel mimarisi, Master ve Worker (Node) olarak adlandırılan iki ana bileşen üzerine kuruludur. İşte bu iki bileşenin rolleri ve işlevleri:

1. Master Node (Master düğümü):

Master Node, Kubernetes kümesinin yönetiminden sorumlu olan ana bileşendir. Genellikle yüksek güvenilirlik ve yüksek performans için birden fazla Master Node kullanılır, ancak genelde üç ya da beş düğüm önerilir. Master Node'un temel bileşenleri şunlardır:

- **API Server:** Kubernetes kümesine istemci tarafından yapılan tüm istekleri kabul eden ve işleyen bileşendir. API Server, kümeyi kontrol etmek için kullanılan tüm operasyonları sağlar.
- **Scheduler:** Yeni oluşturulan pod'ların (konteyner grupları) hangi Worker Node'larda çalışacağını belirler. Scheduler, kaynak kullanımı, policy ve diğer kısıtlamaları göz önünde bulundurarak en uygun Node'u seçer.
- **Controller Manager:** Kümedeki öz durumları (state) izleyen, düzenleyen ve doğrulayan bir arka plan süreçler koleksiyonudur. Örneğin, ReplicaSetController, NodeController, NamespaceController gibi kontrolcüler, belirli nesnelerin istenen durumlarını korur ve sağlar.
- **etcd:** Kubernetes'in tuttuğu tüm kritik bilgilerin (durum bilgisi, yapılandırma verileri vb.) güvenli bir şekilde saklandığı bir dağıtılmış key-value mağazasıdır. etcd, kümenin güvenilirliğini sağlamak için veri tutar ve replike eder.

2. Worker Node (Worker düğümü):

Worker Node, Kubernetes tarafından yönetilen ve çalışan uygulamaların (konteynerlerin) çalıştığı fiziksel veya sanal makinelerdir. Her Worker Node, Kubernetes'in iş yüklerini yürütmek için gerekli olan bileşenleri çalıştırır. Bir Worker Node'un bileşenleri şunlardır:

- **Kubelet:** Master Node'dan gelen komutlarla Worker Node üzerinde pod'ların oluşturulmasını, başlatılmasını ve durumlarının raporlanmasını sağlar. Kubelet, Node üzerinde Kubernetes'in ajanı olarak çalışır.
- **Kube-proxy:** Servislerin ağ geçidi ve yük dengeleyici görevlerini yerine getirir. Kube-proxy, Node üzerindeki ağ yönlendirmesini yönetir ve servislerin girişini dağıtır.
- **Container Runtime:** Konteynerlerin çalıştırılmasını sağlayan alt sistemdir. Örneğin, Docker veya Containerd gibi.

Her Worker Node, belirli kaynaklara (CPU, bellek, depolama vb.) sahip olabilir ve Kubernetes'in etkin kaynak yönetimi sayesinde bu kaynaklar etkin bir şekilde kullanılır. Worker Node'lar, Master Node ile iletişim kurarak yönergeler alır ve

bildirir, böylece Kubernetes kümesindeki uygulamaların dağıtımı, yönetimi ve izlenmesi sağlanır.

Bu şekilde, Master ve Worker Node'lar arasındaki işbirliği sayesinde Kubernetes, yüksek oranda ölçeklenebilir, güvenilir ve yönetilebilir bir konteyner orkestrasyon platformu sunar.

- *Pod Kavramı ve Pod'larda Çalışan Konteynerler.*

Pod, Kubernetes'in temel iş yükü birimidir ve genellikle bir veya daha fazla konteyneri içinde barındırır. Pod, aynı fiziksel veya sanal makine üzerinde çalışan konteynerler grubunu temsil eder ve bu konteynerler birlikte yaşar, paylaştıkları ağ ve depolama alanını kullanabilirler. Pod'un temel özellikleri şunlardır:

Pod Kavramı:

- **Temel İş Yükü Birimi:** Pod, Kubernetes üzerinde dağıtılan en küçük iş yükü birimidir. Bir Pod içinde bir veya birden fazla konteyner bulunabilir.
- **Birlikte Çalışan Konteynerler:** Pod, içindeki konteynerlerin birlikte yaşamasını ve birlikte çalışmasını sağlar. Bu konteynerler, aynı ağ alanını (localhost üzerinden iletişim) ve depolama alanını paylaşabilirler.
- **Pod'a Özgü IP Adresi:** Her Pod, Kubernetes tarafından atanan benzersiz bir IP adresine sahiptir. Konteynerler, bu IP adresi üzerinden dış dünyayla iletişim kurabilirler.
- **Geçici ve Ölçülebilir:** Pod'lar, Kubernetes tarafından ölçeklendirilebilir bir yapıda yönetilir. Birden fazla replikası olabilir ve gerektiğinde artırılabilir veya azaltılabilir.

Pod'larda Çalışan Konteynerler:

- Pod içindeki konteynerler, aynı kaynaklara (CPU, bellek, depolama vb.) erişirler ve aynı Pod yaşam döngüsünü paylaşırlar. Yani, bir Pod'un

oluşturulması, başlatılması, durumu ve sonlandırılması konteynerler arasında ortaktır.

- Tipik bir kullanım senaryosu, bir mikro hizmet mimarisinde her bir Pod'un bir servis için gerekli olan ana uygulama konteynerini ve belki de yardımcı bir yardımcı konteyneri (log toplama, veri tabanı senkronizasyonu gibi) barındırmasıdır.
- Konteynerler, Pod'un kendi namespace'inde izole edilmiş olarak çalışır. Yani, bir Pod içindeki her konteyner, kendi dosya sistemine, ağ alanına ve diğer namespace'lerine sahiptir.

Pod'lar, Kubernetes'in temel yapı taşlarından biri olarak, uygulamalarınızı daha esnek ve yönetilebilir hale getirmenize yardımcı olur. Pod kavramı, konteynerler arasında iletişimi ve işbirliğini kolaylaştırırken, Kubernetes'in diğer özellikleriyle birlikte uygulamalarınızı güvenilir ve ölçeklenebilir bir şekilde dağıtmanızı sağlar.

- Service'ler ve Service Tipleri (ClusterIP , NodePort , LoadBalancer , ExternalName)

Kubernetes'te Service'ler, uygulamalarınızın dış dünyayla iletişim kurmasını sağlayan bir mekanizmadır. Pod'lar dinamik olarak oluşturulup silindiğinde veya farklı Node'larda dağıtıldığında IP adresleri değişebilir. Service'ler, bu IP adresi değişikliklerini yönetir ve uygulamalarınıza sabit bir adres ve DNS adı sağlar.

Service Tipleri:

1. **ClusterIP:**
    - **Tanımı:** ClusterIP, yalnızca Kubernetes içindeki diğer uygulamalarla iletişim kurmak için kullanılır. Bu tip, yalnızca Kubernetes kümesi içindeki diğer Pod'lar tarafından erişilebilir.
    - **Kullanımı:** Genellikle mikro hizmet mimarilerinde servisler arası iletişimde kullanılır. Bu tip servisler, genellikle bir frontend uygulaması ile bağlantı kurmak veya bir backend servisi sağlamak için kullanılır.

2. **NodePort:**
    - **Tanımı:** NodePort, belirli bir port üzerinden tüm Node'lar üzerinden dış dünyaya erişim sağlar. NodePort, ClusterIP'nin sunduğu iç iletişimden daha geniş bir erişim sağlar.
    - **Kullanımı:** Özellikle Kubernetes kümesine dışarıdan erişim gerektiren durumlarda kullanılır. Örneğin, geliştirme veya test aşamalarında uygulamaları dış dünyadan erişilebilir yapmak için NodePort kullanılabilir.
3. **LoadBalancer:**
    - **Tanımı:** LoadBalancer, genellikle bulut sağlayıcılar tarafından sunulan yük dengeleyici (load balancer) servislerini kullanarak dış dünyadan erişimi sağlar. Bu tip, NodePort'un sunduğu erişim imkanının ötesinde, yük dengeleyici aracılığıyla trafik yönlendirme ve dengesini de sağlar.
    - **Kullanımı:** Prodüksiyon ortamlarında ölçeklenebilir ve yüksek erişilebilirlik gereksinimleri olan uygulamalar için idealdir. Örneğin, web uygulamaları veya API servisleri için kullanılır.
4. **ExternalName:**
    - **Tanımı:** ExternalName, Kubernetes içinde bir servis yaratırken, bu servise harici bir DNS adı (externalName) atamak için kullanılır. Bu tip, Kubernetes içindeki uygulamaların harici sistemlere erişimini sağlar.
    - **Kullanımı:** Özellikle harici kaynaklara (örneğin, bir veri tabanı sunucusuna veya bir dış API'ye) erişmek gerektiğinde kullanılır. Servislerin harici DNS adlarıyla bağlantı kurmasını sağlar.

Özetle:

Service'ler, Kubernetes içindeki uygulamaların erişilebilirliğini ve yönetilebilirliğini artıran önemli bir yapı taşıdır. ClusterIP, NodePort, LoadBalancer ve ExternalName gibi farklı Service tipleri, uygulamalarınızın ihtiyaçlarına göre farklı erişim ve iletişim yöntemleri sunarlar, böylece uygulamalarınızın dış dünyayla etkileşimi Kubernetes üzerinde güvenilir bir şekilde yönetilir.

3. Kubernetes Uygulama Yönetimi

- Deployment ve ReplicaSet'ler.

Kubernetes'te uygulama yönetimi yaparken Deployment ve ReplicaSet kavramları önemlidir. İşlevleri ve kullanımları şu şekildedir:

ReplicaSet

ReplicaSet, Kubernetes'in kullanıcı tarafından belirtilen bir sayıda pod'un çalışmasını sağlayan bir yapıdır. Özellikle pod'ların belirli bir sayıda çalışmasını sağlamak ve bu sayıyı korumak için kullanılır. Eğer çalışan pod sayısı istenen sayının altına düşerse veya fazla olursa, ReplicaSet bu durumu tespit eder ve gerekli pod'ları oluşturarak veya kaldırarak istenilen duruma getirir.

Örneğin, bir uygulamanın 3 adet pod'unun her zaman çalışmasını garanti altına almak istiyorsanız, bu 3 pod'u sağlamak üzere bir ReplicaSet tanımlayabilirsiniz. Eğer bir pod çökerse veya başka bir sebeple kapanırsa, ReplicaSet otomatik olarak yeni bir pod oluşturarak 3 pod'u korumaya devam eder.

Deployment

Deployment, ReplicaSet'in üzerine kurulmuş bir daha yüksek seviyeli bir yönetim aracıdır. Deployment, ReplicaSet'in ötesinde uygulamanızın güncellenmesini, sürüm kontrolünü ve geri alma işlemlerini yönetir. Ayrıca, istenmeyen durumlar (örneğin, node'lar veya bölge kesintileri) durumunda pod'ların tekrar oluşturulmasını sağlar.

Deployment'lar genellikle pod şablonunu (template) içerir ve bu şablonu kullanarak ReplicaSet'i yönetir. Yani Deployment, güncelleme yaparken eski ReplicaSet yerine yeni bir ReplicaSet oluşturarak güncelleme işlemini gerçekleştirir ve eski ReplicaSet'i yavaş yavaş kaldırır.

Deployment ve ReplicaSet Arasındaki İlişki

Kısaca özetlemek gerekirse, Deployment, uygulamanızın genel durumunu yönetirken (güncelleme, geri alma gibi), ReplicaSet ise belirli bir pod sayısının sürekli olarak

çalışmasını sağlar. Deployment, ReplicaSet üzerine kurulmuş bir kontrol mekanizması olarak düşünülebilir ve güncellemelerde ReplicaSet'ler arasında geçiş yaparak istenilen durumu korur.

- StatefulSet'ler ve DaemonSet'ler

Kubernetes'te uygulama yönetimi için StatefulSet ve DaemonSet gibi özel kaynak yönetim yapıları önemli roller üstlenir. İşlevleri ve kullanım amaçları aşağıda açıklanmıştır:

StatefulSet

StatefulSet, genellikle durum (state) tutan uygulamalar için kullanılır. Bu tür uygulamalar, her bir örneğin benzersiz bir durum veya veriye sahip olduğu ve aynı zamanda belirli bir sırayla başlatılması veya durdurulması gerektiği durumlarda kullanılır. Örnek olarak veri tabanı sunucuları (MySQL, PostgreSQL gibi) ve mesaj kuyrukları (Kafka, RabbitMQ gibi) bu kategoriye girer.

StatefulSet'in temel özellikleri şunlardır:

- **Sıralı Kimlikler**: Her örneğe benzersiz bir isim ve kimlik atanabilir (örneğin, app-0, app-1, app-2).
- **Kalıcı Depolama**: Pod'ların tuttuğu verilerin kalıcı olması sağlanabilir (örneğin, PersistentVolumes kullanarak).
- **Stable Network Identifiers**: Pod'lar için sabit ağ kimlikleri (hostname) sağlar, böylece diğer pod'lar tarafından kolayca erişilebilirler.

StatefulSet, uygulamaların güvenilir bir şekilde ölçeklenmesini ve yönetilmesini sağlar, ancak uygulama özelleştirmeleri ve veri durumu gibi belirli gereksinimler doğrultusunda daha karmaşık yapılandırmalar gerektirebilir.

DaemonSet

DaemonSet, Kubernetes cluster'ındaki her bir node üzerinde belirli bir pod'ın çalışmasını sağlar. Bu, genellikle cluster'ın her bir node'unda belirli bir servisin (örneğin, log toplama, node ölçümlemesi gibi) çalışmasını gerektiren durumlarda kullanılır.

DaemonSet'in temel özellikleri şunlardır:

- **Her Node'da Bir**: Her node'da bir pod olacak şekilde çalışır. Yeni node'lar eklenirse, DaemonSet otomatik olarak bu node'lara pod'ları dağıtır.
- **Node Spesifik Ayarlar**: Node'lar arasında farklı yapılandırmalar veya ayarlar gerektiren durumlar için uygundur.
- **Cluster Genel Servisler**: Cluster'ın tamamında belirli bir görevi (servisi) dağıtmak için idealdir.

DaemonSet'ler genellikle alt yapı bileşenleri ve çeşitli arka plan işlemleri için kullanılır. Örneğin, log toplama araçları için DaemonSet kullanarak her node'da bir log toplama agent'ı çalıştırılabilir.

- Pod Yaşam Döngüsü: Oluşturma , Güncelleme , Silme.

Kubernetes'te bir Pod'un yaşam döngüsü, oluşturma, güncelleme ve silme aşamalarından geçer. Her bir aşama, Kubernetes'in kaynak yönetimi ve operasyonel süreçlerinin bir parçası olarak önemlidir. İşte Pod'un yaşam döngüsü adımları:

1. Oluşturma (Creation)

Pod oluşturma süreci, kullanıcı tarafından veya bir controller (örneğin, Deployment, StatefulSet gibi) tarafından başlatılabilir. Oluşturma süreci genellikle şu adımları içerir:

- **API Sunucusuna İstek Gönderme**: Kullanıcı veya controller, yeni bir Pod'un oluşturulmasını isteyen bir API isteği gönderir.

- **Scheduling (Planlama)**: Kubernetes, Pod'un nerede çalıştırılacağına karar verir (hangi node üzerinde çalışacak) ve Pod'un spesifikasyonuna uygun bir node belirler.
- **Pod Oluşturma**: Belirlenen node üzerinde Kubernetes, Pod'u oluşturmak için gerekli konteynerleri başlatır. Pod'un spesifikasyonunda belirtilen konteynerler, volume'lar ve diğer özellikler bu aşamada yapılandırılır ve başlatılır.
- **Pod Durumunu Kontrol Etme**: Kubernetes, oluşturulan Pod'un durumunu izler ve gerekirse durumunu günceller.

2. Güncelleme (Update)

Pod güncelleme süreci, genellikle Pod'un spesifikasyonundaki değişiklikler veya yeni bir versiyonun devreye alınması gerektiğinde gerçekleşir. Güncelleme aşamaları şunları içerebilir:

- **Yeni Konfigürasyonu Uygulama**: Pod spesifikasyonunda yapılan değişiklikler (örneğin, konteyner imajı veya ortam değişkenleri güncellemeleri) API üzerinden gönderilir.
- **Rolling Update (Yavaş Güncelleme)**: Kubernetes, güncellenmiş Pod spesifikasyonuna göre yeni Pod'ları sırayla devreye alır ve eski Pod'ları kaldırır. Bu şekilde, uygulama kesintiye uğramadan güncellenir.
- **Güncelleme Durumunu İzleme**: Kubernetes, güncelleme işleminin durumunu izler ve gerektiğinde geri alma veya hata yönetimi işlemlerini uygular.

3. Silme (Deletion)

Pod silme süreci, Pod'un artık gerekli olmadığı veya kaldırılması gerektiği zaman gerçekleşir. Silme süreci şu şekilde ilerler:

- **Silme İsteği Gönderme**: Kullanıcı veya controller, Pod'u silmek için API üzerinden bir istek gönderir.

- **Silme Politikalarını Uygulama**: Kubernetes, Pod'u silmeden önce belirli politikaları uygular. Örneğin, Pod'un hemen durdurulması veya son işlemlerinin tamamlanması için beklenmesi gibi.
- **Pod'u Kaldırma**: Kubernetes, Pod'u çalışan node'dan kaldırır ve kaynaklarını serbest bırakır.
- **Silme Durumunu İzleme**: Kubernetes, Pod'un başarıyla silinip silinmediğini izler ve gerekirse hata yönetimi yapar.

Pod'un yaşam döngüsü bu adımları takip eder ve bu süreçler, Kubernetes'in otomatikleştirilmiş, yönetilebilir ve güvenilir bir altyapı sunmasını sağlar.

4. Kubernetes Kaynak Yönetimi

- *CPU ve bellek yönetimi: Requests ve Limits*

Kubernetes'te CPU ve bellek yönetimi, requests ve limits adı verilen iki önemli kavram üzerinden yapılmaktadır. Bu kavramlar, Kubernetes cluster'ında çalışan pod'ların kaynak kullanımını optimize etmek ve dengeli bir şekilde dağıtmak için kullanılır.

Requests (İstekler)

Requests (İstekler), bir pod'un çalışması için Kubernetes cluster'ında rezerve edilen minimum CPU ve bellek miktarını ifade eder. Yani, pod'un başlaması ve çalışması için asgari gereksinimleri belirtir.

- **CPU Requests**: Pod'un bir node üzerinde minimum ne kadar CPU kaynağı kullanacağını belirtir. Örneğin, 100m (0.1 CPU çekirdeği) gibi ifade edilir.
- **Bellek Requests**: Pod'un bir node üzerinde minimum ne kadar bellek kullanacağını belirtir. Örneğin, 256Mi (256 Mebibyte) gibi ifade edilir.

Bu istekler, Kubernetes scheduler'ı tarafından kullanılırken, pod'un hangi node'a yerleştirileceğini ve kaynakların nasıl tahsis edileceğini belirlemekte kullanılır.

Limits (Sınırlar)

Limits (Sınırlar), bir pod'un maksimum CPU ve bellek kullanımını belirler. Yani, pod'un aşırı kaynak kullanımı durumunda sistem tarafından uygulanacak en üst sınırdır.

- **CPU Limits**: Pod'un maksimum ne kadar CPU kaynağı kullanabileceğini belirtir. Örneğin, 500m (0.5 CPU çekirdeği) gibi ifade edilir.
- **Bellek Limits**: Pod'un maksimum ne kadar bellek kullanabileceğini belirtir. Örneğin, 1Gi (1 Gibibyte) gibi ifade edilir.

Limits belirtilmezse, pod kullanılabilir tüm kaynakları kullanabilir. Limits belirlenmesi, diğer pod'ların ve Kubernetes cluster'ının genel performansını korumak ve istikrarını sağlamak için önemlidir.

- Horizontal Pod Autoscaler (HPA) ve Vertical Pod Autoscaler (VPA).

Kubernetes'te kaynak yönetimi ve otomatik ölçeklendirme için iki önemli kavram olan Horizontal Pod Autoscaler (HPA) ve Vertical Pod Autoscaler (VPA) vardır. Her ikisi de farklı senaryolarda kullanılarak uygulamaların performansını ve verimliliğini artırmaya yardımcı olur.

Horizontal Pod Autoscaler (HPA)

Horizontal Pod Autoscaler (HPA), bir uygulamanın pod sayısını otomatik olarak ölçeklendirmek için kullanılan bir Kubernetes özelliğidir. HPA, uygulamanın yüküne göre pod sayısını artırıp azaltarak, talebi karşılayacak şekilde kaynakları dinamik olarak ayarlar.

HPA'nın temel özellikleri şunlardır:

- **Hedef CPU Kullanımı**: HPA, pod'ların CPU kullanımını izleyerek, belirli bir hedef değere (CPU yüzdesi olarak) göre pod sayısını otomatik olarak ayarlar.

- **Ölçekleme Politikaları**: Kullanıcılar, HPA objesini oluştururken belirli ölçekleme politikaları (minimum, maksimum pod sayısı) belirtebilirler.

- **Metril Kaynakları**: HPA, CPU kullanımı yanı sıra özelleştirilmiş metrikler veya hafıza kullanımı gibi diğer metrikleri de temel alarak ölçeklendirme yapabilir.

Vertical Pod Autoscaler (VPA)

Vertical Pod Autoscaler (VPA), pod'ların içinde çalıştığı konteynerlerin CPU ve bellek ihtiyaçlarını dinamik olarak ölçeklendirme amacıyla kullanılır. VPA, pod'ların CPU ve bellek ihtiyaçlarını belirler ve bunları dinamik olarak ayarlayarak, kaynak kullanımını optimize eder.

VPA'nın temel özellikleri şunlardır:

- **Kaynak İhtiyaçlarını Belirleme**: VPA, pod'ların çalışma zamanında CPU ve bellek ihtiyaçlarını belirleyerek, gerektiğinde bu kaynakları artırır veya azaltır.

- **Otamatik Yeniden Başlatma**: VPA, pod'ların kaynaklarını güncellediğinde, pod'ları otomatik olarak yeniden başlatarak güncellemelerin etkili olmasını sağlar.

- **Cluster Genelinde Optimizasyon**: VPA, Kubernetes cluster'ındaki kaynak kullanımını optimize etmek için kullanılabilir ve ölçeklenebilir bir çözüm sunar.

5. Kubernetes Depolama Yönetimi

- *Volume ve Volume Tipleri (emptyDir , hostPath , PersistentVolume Claim)*

Kubernetes'te depolama yönetimi, Volume ve çeşitli Volume Tipleri üzerinden

yapılandırılır. Her bir volume tipi farklı kullanım senaryoları için tasarlanmıştır ve farklı özelliklere sahiptir. İşte yaygın olarak kullanılan volume tipleri:

1. emptyDir

emptyDir, geçici ve pod ömrü boyunca yaşayan bir volume tipidir. Pod oluşturulduğunda oluşturulur ve pod silindiğinde veya yeniden başlatıldığında içeriği silinir. Temel özellikleri şunlardır:

- **Kullanım Senaryoları**: Geçici veri depolama veya pod'lar arasında geçici veri paylaşımı gibi senaryolarda kullanılır.

- **Özellikler**: Pod içindeki tüm konteynerler tarafından erişilebilir. Pod yeniden başlatıldığında içeriği sıfırlanır.

Örnek bir ‘emptyDir’ volume kullanımı:

```yaml
volumes:
  - name: temp-data
    emptyDir: {}
```

2. hostPath

hostPath, bir node'un dosya sistemindeki belirli bir yolu pod içinde bir volume olarak monte etmek için kullanılır. Bu volume tipi, pod'ların node'a bağlı olduğu özel durumlarda ve cluster dışı depolama ihtiyaçları için kullanılır.

- **Kullanım Senaryoları**: Node spesifik kaynaklara erişim gerektiren uygulamalar veya cluster dışı kaynaklara ihtiyaç duyan uygulamalar için uygundur.

- **Özellikler**: Pod'a özgüdür ve pod silindiğinde veya node yeniden başlatıldığında veri kaybolabilir. Güvenlik ve izolasyon riski taşıyabilir.

Örnek bir ‘hostPath’ volume kullanımı:

```yaml
volumes:
  - name: host-data
    hostPath:
      path: /var/data
```

3. PersistentVolumeClaim (PVC)

PersistentVolumeClaim (PVC), Kubernetes'te kalıcı depolama çözümlerine erişmek için kullanılan bir mekanizmadır. PVC, PersistentVolume'ların (PV) dinamik olarak talep edilmesini sağlar ve pod'lar arasında veri saklamak için kullanılır.

- **Kullanım Senaryoları**: Uygulama verilerinin kalıcı olarak saklanması gereken durumlar için kullanılır. Örneğin, veritabanı depolama, dosya depolama gibi.
- **Özellikler**: PV'lerle dinamik olarak eşleştirilebilir ve farklı depolama sınıflarına (storage class) göre ayarlanabilir. Veriler pod'un ömrü boyunca kalıcıdır.

Örnek bir ‘PersistentVolumeClaim’ tanımı:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-data-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Bu örnek, 1Gi boyutunda bir depolama alanı talep eden bir PVC tanımlar. PVC, PVC'nin belirli bir sınıftan (örneğin, SSD veya HDD) dinamik olarak tahsis edilmesini sağlar.

Özet

- **emptyDir**: Geçici veri depolama için, pod ömrü boyunca geçerli.
- **hostPath**: Node'a özgü dosya sistemine erişim için, güvenlik ve izolasyon riskleri olabilir.
- **PersistentVolumeClaim (PVC)**: Kalıcı veri depolama için, pod'lar arası veri saklamak ve kalıcı depolama alanı talep etmek için kullanılır.

Bu volume tipleri, farklı uygulama gereksinimlerini karşılamak üzere Kubernetes içinde kullanılır ve her biri kendi avantajları ve dezavantajlarına sahiptir. Uygulamanın ihtiyacına göre doğru volume tipinin seçilmesi önemlidir.

- StorageClass'lar ve PersistentVolume'lar.

Kubernetes'te depolama yönetimi için StorageClass ve PersistentVolume (PV) kavramları önemlidir. Bu kavramlar, uygulamaların ihtiyaç duyduğu kalıcı depolama çözümlerini dinamik olarak sağlamak ve yönetmek için kullanılır.

StorageClass

StorageClass, Kubernetes'te dinamik olarak oluşturulabilen ve PVC'ler tarafından talep edilen kalıcı depolama sınıflarını tanımlayan bir yapıdır. Her StorageClass, belirli depolama sağlayıcıları veya özelliklerle ilişkilendirilmiş olabilir. StorageClass'lar aşağıdaki amaçlar için kullanılır:

- **Depolama Sınıfları Tanımlama**: Farklı depolama sınıflarını (örneğin, SSD, HDD gibi) tanımlamak ve belirli depolama sağlayıcıları veya özellikleri ayarlamak için kullanılır.
- **Depolama Dinamik Tahsisi**: PVC'ler, StorageClass kullanarak dinamik olarak PV (PersistentVolume) tahsis edebilir. Bu, yöneticinin elle PV oluşturmasını gerektirmez.

Örnek bir StorageClass tanımı:

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
```

Bu örnekte, ‘fast’ adında bir StorageClass tanımlanmıştır. Bu StorageClass, AWS EBS (Elastic Block Store) ile entegre edilmiş ve ‘gp2’ türünde SSD depolama sağlar.

PersistentVolume (PV)

PersistentVolume (PV), Kubernetes cluster'ında, depolama sınıflarıyla ilişkilendirilmiş ve pod'lar arasında paylaşılabilen bir depolama kaynağını temsil eder. PV'ler genellikle cluster yöneticisi tarafından elle veya dinamik olarak sağlanabilir ve PVC'ler tarafından talep edilir.

- **PV Tanımlama**: PV, fiziksel veya bulut tabanlı depolama sağlayıcılarından (AWS EBS, GCP Persistent Disk, Azure Disk gibi) gelen depolama alanını temsil eder.

- **Depolama Sınıflarıyla İlişkilendirme**: PV, bir veya birden fazla StorageClass ile ilişkilendirilebilir ve bu sayede PVC'ler tarafından talep edilebilir.

Örnek bir PV tanımı:

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-example
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: fast
  awsElasticBlockStore:
    volumeID: "vol-1234567890abcdef0"
    fsType: ext4
```

Bu örnekte, ‘pv-example’ adında bir PV tanımlanmıştır. PV, 1Gi boyutunda AWS EBS (Elastic Block Store) cihazı ile oluşturulmuş ve ‘fast’ StorageClass ile ilişkilendirilmiştir. PV'nin kullanım modu (‘volumeMode’), erişim modları (‘accessModes’), geri alma politikası (‘persistentVolumeReclaimPolicy’) gibi özellikleri de tanımlanmıştır.

Özet

- **StorageClass**: PVC'lerin dinamik olarak PV talep etmesini sağlayan ve belirli depolama sağlayıcıları veya özelliklerle ilişkilendirilmiş depolama sınıflarını tanımlayan Kubernetes objesi.
- **PersistentVolume (PV)**: Kubernetes cluster'ında depolama sağlayıcılarından gelen ve belirli erişim modları ve özelliklerle ilişkilendirilmiş kalıcı depolama kaynağını temsil eden obje.

Bu yapılar, Kubernetes cluster'ında uygulamaların kalıcı depolama ihtiyaçlarını yönetmek ve dinamik olarak tahsis etmek için kullanılır. StorageClass'lar ve PV'lerin doğru şekilde yapılandırılması, uygulamaların güvenilir ve performanslı bir şekilde çalışmasını sağlar.

6. Kubernetes Ağ Yönetimi

- Service Networking (Cluster DNS , Service Discovery)

Kubernetes'te hizmet ağı (service networking), uygulamalar arasında iletişimi sağlamak ve hizmetlerin birbirlerini keşfetmelerini mümkün kılmak için önemli bir yapılandırmadır. Bu kavramlar genellikle Cluster DNS ve hizmet keşfi (Service Discovery) ile ilişkilendirilir.

Cluster DNS

Cluster DNS, Kubernetes cluster'ında çalışan hizmetlerin ve pod'ların IP adreslerini çözümlemek için kullanılan DNS (Domain Name System) servisidir. Her Kubernetes cluster'ında, Kubernetes tarafından otomatik olarak yönetilen bir DNS servisi bulunur. Bu servis, pod'ların ve hizmetlerin Kubernetes içindeki FQDN'lerini (Fully Qualified Domain Name) ve IP adreslerini çözebilmesini sağlar.

- **Kullanım Senaryoları**: Pod'lar, diğer pod'lar veya hizmetlerle iletişim kurarken, IP adresi yerine FQDN kullanarak iletişim kurabilirler. Bu, uygulamalar arasında esnek ve dinamik iletişimi sağlar.
- **Özellikler**: Kubernetes cluster'ında her pod ve hizmet için otomatik olarak DNS kayıtları oluşturulur ve güncellenir. Bu sayede hizmetler dinamik olarak ölçeklendirilebilir ve DNS çözümlemesi otomatik olarak güncellenir.

Örneğin, bir pod başka bir pod veya hizmet ile iletişim kurarken, Kubernetes cluster'ındaki Cluster DNS kullanılır ve pod'ların IP adreslerini FQDN olarak çözümler.

Service Discovery (Hizmet Keşfi)

Service Discovery, Kubernetes cluster'ındaki hizmetlerin dinamik olarak keşfedilmesi ve erişilmesi için kullanılan bir mekanizmadır. Kubernetes, Service objeleri aracılığıyla bu keşif işlemini yönetir. Service objeleri, aynı hizmetin birden fazla pod örneğini tek bir sanal IP ve FQDN altında gruplandırarak sunar.

- **Kullanım Senaryoları**: Uygulamalar, diğer hizmetlerle iletişim kurarken Service objelerini hedef alarak, Kubernetes tarafından yönetilen hizmetlerin pod'larını dinamik olarak keşfedebilirler.

- **Özellikler**: Service objeleri, Kubernetes tarafından otomatik olarak load balancing ve failover sağlar. Bu sayede uygulamaların hizmetlere güvenilir ve skalabilir bir şekilde erişmesi sağlanır.

Örneğin, bir uygulama başka bir hizmete HTTP isteği göndermek istediğinde, ilgili Service objesinin FQDN'sini hedef alır ve Kubernetes tarafından yönetilen hizmetin tüm pod örnekleri arasında load balancing yaparak isteği yönlendirir.

Özet

- **Cluster DNS**: Kubernetes cluster'ındaki pod'ların ve hizmetlerin IP adreslerini ve FQDN'lerini çözümlemek için kullanılan DNS servisi.

- **Service Discovery**: Kubernetes cluster'ındaki hizmetlerin dinamik olarak keşfedilmesi ve erişilmesi için kullanılan mekanizma, Service objeleri aracılığıyla sağlanır.

Bu yapılar, Kubernetes cluster'ında uygulamalar arası iletişimi yönetmek ve hizmetlerin dinamik olarak ölçeklenmesini sağlamak için kritik öneme sahiptir. Cluster DNS ve Service Discovery mekanizmaları, Kubernetes'in esnek ve ölçeklenebilir hizmet yönetimi sağlayabilmesinin temel taşlarından biridir.

-Pod Networking (CNI , Container Network Interface)

Kubernetes'te pod ağı (Pod Networking) kavramı, pod'ların birbiriyle ve dış dünyayla iletişim kurabilmesi için gereken ağ konfigürasyonunu ve yönetimini içerir. Bu kavram, Kubernetes cluster'ında çalışan pod'ların ağ üzerinde nasıl iletişim kurduğunu ve ağ kaynaklarını nasıl kullandığını tanımlar. İki önemli bileşen bu bağlamda öne çıkar: Container Network Interface (CNI) ve pod ağı yapılandırması.

Container Network Interface (CNI)

Container Network Interface (CNI), Kubernetes ve diğer container orchestration sistemleri tarafından kullanılan standart bir API ve arayüzdür. CNI, container runtime (örneğin Docker, containerd) tarafından kontrol edilen ağ bağlantılarını yönetir ve pod'ların ağ bağlantılarını nasıl yapılandırılacağını belirler.

- **Kullanım Senaryoları**: Pod'ların ağ konfigürasyonunun yapılması, IP adreslerinin atanması, ağ izolasyonu ve güvenlik politikalarının uygulanması gibi görevleri CNI sağlar.

- **Özellikler**: CNI, birden fazla ağ sağlayıcısı (network plugin) ile entegre edilebilir. Bu plugin'ler, farklı ağ modellemeleri (overlay networks, L3 routing, L2 bridging) sunabilir ve Kubernetes cluster'ında pod'ların iletişimini yönetir.

Kubernetes cluster'ında CNI kullanımı, genellikle cluster'ın ağ mimarisine ve kullanılan ağ plugin'lerine (örneğin, Calico, Flannel, Weave gibi) bağlıdır. Örneğin, Calico CNI, Kubernetes için popüler bir ağ plugin'idir ve L3 routing ile ağ segmentasyonunu ve güvenliğini sağlar.

Pod Ağı Yapılandırması

Kubernetes'te pod ağı yapılandırması, pod'ların ağ IP adreslerinin atanması, ağ trafiğinin yönlendirilmesi ve ağ politikalarının uygulanması gibi işlemleri içerir. Pod'ların ağ yapılandırması aşağıdaki özelliklere sahiptir:

- **IP Atama**: Her pod'a benzersiz bir IP adresi atanır. Bu IP adresi pod'un yaşam döngüsü boyunca sabit kalır ve pod'u ağ üzerinde benzersiz bir şekilde tanımlar.

- **Pod-to-Pod İletişim**: Kubernetes cluster'ındaki pod'lar, herhangi bir node üzerindeki diğer pod'larla doğrudan iletişim kurabilir. Bu iletişim, CNI tarafından sağlanan ağ yapılandırması üzerinden gerçekleşir.

- **Service İletişimi**: Service objeleri, pod'ların birbiriyle veya dış dünya ile iletişim kurabilmesi için bir Service IP ve FQDN (Fully Qualified Domain Name) sağlar. Service objeleri, pod'ları gruplayarak load balancing ve pod yeniden yönlendirme işlevleri de sağlar.

Özet

- **Container Network Interface (CNI)**: Kubernetes cluster'ında pod'ların ağ bağlantılarını yöneten standart API ve arayüz.

- **Pod Ağı Yapılandırması**: Pod'ların IP adresi atanması, pod-to-pod iletişimi, service objeleri aracılığıyla hizmet keşfi ve iletişimi gibi pod ağı üzerindeki temel yapılandırma ve yönetim işlemleri.

Bu yapılar, Kubernetes cluster'ında çalışan uygulamaların ağ üzerinde güvenli, performanslı ve ölçeklenebilir bir şekilde iletişim kurabilmesini sağlar. CNI ve pod ağı yapılandırması, Kubernetes'in esnek ve dağıtık ağ yönetimi yeteneklerini destekler.

-Ingress ve Ingress Controllers

Kubernetes'te Ingress ve Ingress Controllers, uygulamaların dış dünyaya erişimini yönetmek ve HTTP/HTTPS trafiğini belirli kurallara göre yönlendirmek için kullanılan önemli kavramlardır.

Ingress

Ingress, Kubernetes cluster'ında dış dünyadan gelen HTTP/HTTPS trafiğini yönetmek için bir API objesidir. Ingress, cluster'daki bir Service'e gelen trafik için yönlendirme kuralları ve güvenlik ayarlarını tanımlar.

- **Kullanım Senaryoları**: Ingress, farklı domainlere veya URL yollarına gelen trafikleri farklı hizmetlere yönlendirme, SSL/TLS şifrelemesi ve yük dengeleme gibi özellikler sunar.
- **Özellikler**: Ingress, Layer 7 (HTTP) trafiğini yönetir ve bu sayede URL tabanlı yönlendirme ve SSL termination gibi gelişmiş HTTP işlevselliği sağlar

Ingress Controller

Ingress Controller, Ingress objelerini okuyan ve onları cluster'da uygulayan bir Kubernetes bileşenidir. Ingress Controller, cluster içindeki HTTP trafiklerini Ingress kurallarına göre yönlendirir ve uygular.

- **Kullanım Senaryoları**: Kubernetes cluster'ında Ingress Controller, belirli bir Ingress objesini izleyerek, Service'ler arasında trafik yönlendirme kurallarını uygular. Farklı Ingress Controller'lar farklı özellikler sunabilir (örneğin, Nginx Ingress Controller, Traefik gibi).
- **Özellikler**: Ingress Controller, genellikle yük dengeleme, SSL/TLS şifrelemesi, URL yönlendirme gibi gelişmiş HTTP işlevselliği sağlar. Ayrıca, dinamik olarak Ingress kurallarını güncelleyebilir ve yönetebilir.

Kubernetes cluster'ında birçok Ingress Controller seçeneği bulunmaktadır ve her biri farklı özellikler ve performans profilleri sunabilir. Örneğin, Nginx veya Traefik gibi popüler Ingress Controller'lar, genişletilebilirlik ve özelleştirilebilirlik sağlar.

Özet

- **Ingress**: Kubernetes cluster'ında HTTP/HTTPS trafiğini yönetmek için kullanılan bir API objesi. URL tabanlı yönlendirme, SSL/TLS şifrelemesi gibi özellikler sunar.
- **Ingress Controller**: Ingress objelerini okuyan ve uygulayan Kubernetes bileşeni. Ingress kurallarına göre HTTP trafiğini yönlendirir ve gerekli işlevselliği sağlar.

Bu yapılar, Kubernetes cluster'ında uygulamaların dış dünyaya güvenli ve etkin bir şekilde erişmesini sağlar. Ingress ve Ingress Controller'lar, ölçeklenebilir ve esnek HTTP trafiği yönetimi için önemli araçlardır.

7. Kubernetes Güvenlik

- RBAC (Role-Based Access Control) ve Service Accounts

Kubernetes'te güvenlik yönetimi için kullanılan iki önemli kavram RBAC (Role-Based Access Control) ve Service Accountsdir. Bu kavramlar, Kubernetes cluster'ında hangi kullanıcıların hangi kaynaklara erişebileceğini ve hangi izinlere sahip olabileceğini tanımlamak için kullanılır.

Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC), Kubernetes'te kullanıcıların veya işlemlerin cluster içindeki kaynaklara erişimini yönetmek için kullanılan bir güvenlik modelidir. RBAC, üç temel kavram üzerine kuruludur: Roller (Roles), Rollerin Bağlanması (Role Bindings) ve Kullanıcılar (Users) veya Servis Hesapları (Service Accounts).

- **Kullanım Senaryoları**: RBAC, cluster içinde farklı kullanıcıların veya grupların belirli kaynaklara erişimini kontrol etmek için kullanılır. Örneğin, bir

kullanıcının sadece belirli namespace içindeki pod'ları görebilmesi veya bir grupun sadece belirli bir deployment'ı güncelleyebilmesi gibi.

- **Özellikler**: RBAC, granüler izinler sağlar ve Kubernetes objelerine (pod'lar, service'ler, configMap'ler vb.) erişim yetkilerini yönetir. Cluster yöneticileri, RBAC rolleri ve rollerin bağlanmasını yöneterek güvenlik politikalarını uygular.

  Örnek bir RBAC rolü ve rol bağlantısı tanımı:

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]

---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Bu örnekte, pod-reader adında bir RBAC rolü tanımlanmış ve get ve list gibi işlemleri yapmak için pods kaynaklarına izin verilmiştir. alice adında bir kullanıcıya (User) bu rol, read-pods adında bir RoleBinding ile bağlanmıştır.

Service Accounts

Service Accounts, Kubernetes içinde çalışan pod'ların ve uygulamaların diğer Kubernetes objeleri ile (API sunucusu gibi) etkileşimde bulunabilmesi için kullanılan hesaplardır.

- **Kullanım Senaryoları**: Pod'lar, Kubernetes API sunucusu veya diğer Kubernetes bileşenleri ile iletişim kurarken, Service Account kullanılır. Bu hesaplar, pod'ların kimlik doğrulamasını ve yetkilendirilmesini sağlar.
- **Özellikler**: Her pod, varsayılan olarak bir Service Account ile ilişkilendirilir. Bu hesaplar, pod'un cluster içindeki diğer kaynaklara erişim yetkilerini belirler ve RBAC rollerine göre izinleri kontrol eder.

Örnek bir Service Account tanımı:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
  namespace: default
```

Bu örnekte, ‘default’ namespace içinde ‘my-service-account’ adında bir Service Account tanımlanmıştır. Bu Service Account, pod'lar tarafından kullanılabilir ve RBAC rolleriyle ilişkilendirilebilir.

Özet

- **Role-Based Access Control (RBAC)**: Kubernetes'te kaynaklara erişim kontrolü sağlayan güvenlik modeli. Roller (Roles), Rollerin Bağlanması (Role Bindings) ve Kullanıcılar (Users) veya Servis Hesapları (Service Accounts) üzerine kuruludur.
- **Service Accounts**: Kubernetes pod'larının Kubernetes API sunucusu veya diğer bileşenlerle etkileşimde bulunabilmesi için kullanılan hesaplar.

Bu yapılar, Kubernetes cluster'ında güvenlik politikalarını uygulamak ve kaynaklara erişimi kontrol etmek için önemli araçlardır. RBAC ve Service Accounts, Kubernetes'in güvenli ve izole çalışmasını sağlayan temel bileşenlerdir.

- Network Policies

Network Policies, Kubernetes'te pod'ların ağ trafiğini yönetmek için kullanılan bir güvenlik mekanizmasıdır. Bu politikalar, hangi pod'ların birbirleriyle ve dış dünya ile iletişim kurabileceğini, hangi protokollerin ve portların kullanılabileceğini belirler.

Kullanım Senaryoları

Network Policies genellikle aşağıdaki senaryolar için kullanılır:

1. **Ağ Güvenliği**: Belirli pod'ların sadece belirli diğer pod'larla iletişim kurmasını sağlar. Örneğin, bir database pod'unun sadece uygulama pod'ları tarafından erişilebilmesi.
2. **Segmentasyon**: Farklı uygulama veya hizmetler arasında ağ trafiği segmentasyonu sağlar. Örneğin, geliştirme ve üretim ortamları arasında trafiği izole etmek.
3. **Dış İletişim Kontrolü**: Pod'ların dış dünya ile iletişimini kısıtlamak veya belirli protokol ve portlara erişimi kontrol etmek.

Özellikler

- **Etiket Tabanlı Yönetim**: Network Policies, pod'lara uygulanan etiketlerle ilişkilendirilir. Bu sayede belirli gruplara (örneğin, app: frontend, env: production gibi) ağ politikaları tanımlanabilir.
- **Kapsamlı Kontrol**: Ağ politikaları, pod'ların giriş ve çıkış trafiklerini ve bu trafiklerin kaynakları (IP adresleri, portlar, protokoller) üzerinde detaylı kontrol sağlar.

- ***TSL/SSL kullanımı ve güvenlik en iyi uygulamaları***

Transport Layer Security (TLS) ve Secure Sockets Layer (SSL), ağ iletişiminde güvenliği sağlamak için kullanılan protokollerdir. TLS, SSL'in geliştirilmiş ve daha güvenli bir sürümüdür ve genellikle SSL terimi, TLS ile eşanlamlı olarak kullanılmaktadır. Kubernetes ortamlarında TLS/SSL kullanımı ve güvenlik en iyi uygulamaları aşağıdaki temel noktalardan oluşur:

1. SSL/TLS Sertifikaları

- **Güvenilir Sertifikalar Kullanın**: Sunucu sertifikalarınızın güvenilir bir CA (Certificate Authority) tarafından imzalanmış olmasını sağlayın. Kendi CA'nızı kullanıyorsanız, root sertifikasını güvenli bir şekilde dağıtın ve güncel tutun.

- **Geçerlilik Sürelerini ve Yenilemeleri İzleyin**: Sertifikalarınızın geçerlilik sürelerini düzenli olarak kontrol edin ve süresi dolan sertifikaları zamanında yenileyin.

2. Kubernetes ve SSL/TLS

- **Ingress ve Load Balancer Konfigürasyonu**: Ingress Controller veya Load Balancer gibi Kubernetes kaynaklarında SSL/TLS terminasyonu sağlayın. Bu, dışarıdan gelen HTTPS trafiğini pod'lara yönlendirirken şifrelemenin cluster dışında sonlanmasını sağlar.

- **Pod Noktaları ve İletişim**: Pod'lar arasındaki iç trafiği de SSL/TLS ile şifreleyin. Pod-to-pod iletişimde ağ politikaları ve doğru ağ yapılandırması (örneğin, Calico) kullanarak güvenliği artırın.

Özet

Kubernetes ortamlarında SSL/TLS kullanımı ve güvenlik en iyi uygulamaları, hem dışarıdan gelen trafiği hem de pod'lar arasındaki iletişimi korumak için kritik öneme sahiptir. SSL/TLS sertifikalarını doğru şekilde yönetmek, güvenlik politikalarını doğru uygulamak ve sürekli olarak izlemek, Kubernetes cluster'ınızın güvenliğini artırmanın anahtarıdır. Bu en iyi uygulamaları takip ederek, potansiyel güvenlik açıklarını en aza indirgeyebilir ve veri bütünlüğünü sağlayabilirsiniz.

8 .Kubernetes İzleme ve Günlük Yönetimi

- *Kubernetes bileşenleri için günlük yönetimi*

Kubernetes bileşenleri için günlük yönetimi, cluster'ın sağlığını izlemek, sorunları tespit etmek ve operasyonel verimliliği artırmak için önemli bir süreçtir. İşte Kubernetes bileşenleri için günlük yönetimiyle ilgili bazı önemli noktalar:

1. Kubernetes Bileşenleri ve Günlükler

Kubernetes cluster'ı, birçok farklı bileşenden oluşur ve her bir bileşenin kendi günlükleri vardır. Önemli bileşenler şunları içerir:

- **kube-apiserver**: Kubernetes API'sine gelen istekleri işler.
- **kube-controller-manager**: Kubernetes controller'larını çalıştırır.
- **kube-scheduler**: Pod'ların hangi node'larda çalışacağını seçer.
- **kubelet**: Her bir node üzerindeki pod'ları yönetir.
- **kube-proxy**: Service'ler için ağ yönlendirme ve load balancing yapar.
- **etcd**: Kubernetes cluster'ının durumunu saklayan dağıtılmış bir key-value store.

2. Günlük Toplama ve Merkezi Log Yönetimi

- **Günlük Formatı ve Kaydı**: Her bileşenin günlük formatı ve içeriği farklı olabilir. Örneğin, kube-apiserver günlükleri genellikle API istekleri, kimlik doğrulama durumları gibi bilgileri içerir.
- **Merkezi Log Toplama**: Kubernetes cluster'ındaki tüm bileşenlerden günlükleri merkezi bir yerde toplamak önemlidir. Bu, sorunları tespit etmek ve hızlı bir şekilde yanıt vermek için kritik öneme sahiptir.
- **Log Toplama Araçları**: Elasticsearch, Fluentd, Logstash (ELK Stack), Splunk gibi araçlar, Kubernetes günlüklerini toplamak, depolamak ve analiz etmek için yaygın olarak kullanılır.

3. Günlük Düzeyleri ve Ayarlamaları

- **Günlük Seviyeleri**: Her bileşen için uygun günlük seviyelerini ayarlamak önemlidir. Debug seviyesindeki günlükler, detaylı sorun giderme için yararlı olabilirken, Production ortamlarında genellikle daha düşük seviyeler tercih edilir.
- **Rotasyon ve Temizleme**: Günlük dosyalarının rotasyonu ve gereksiz verilerin temizlenmesi, depolama alanını yönetmek ve performansı korumak için önemlidir.

4. Günlükleri İzleme ve Alarm

- **Günlük İzleme**: Kubernetes bileşenlerinin günlüklerini izlemek için otomatik araçlar kullanın. Örneğin, Prometheus ve Grafana gibi araçlar, günlük verilerini gerçek zamanlı olarak izlemenize ve ölçütler belirlemenize yardımcı olabilir.
- **Uyarılar**: Kritik hataları ve uyarıları algılamak için günlük izleme araçlarına uyarılar ekleyin. Örneğin, belirli bir hata sıklığı aşıldığında veya belirli bir bileşen durduğunda e-posta veya anlık mesaj gönderimi gibi.

5. Günlük Analizi ve Sorun Giderme

- **Günlük Analizi**: Günlüklerden anlamlı bilgiler elde etmek için analiz yapın. Örneğin, CPU veya bellek sınırlarını aşan pod'ları tespit edin veya ağ ile ilgili sorunları belirleyin.

- **Sorun Giderme**: Sorunları tespit etmek ve çözmek için günlük analizlerini kullanın. Örneğin, bir pod'un neden başlatılamadığını veya bir servisin neden erişilemediğini anlamak için günlüklerin detaylı incelemesini yapın.

6. Olay Günlükleri (Event Logs)

- **Kubernetes Olay Günlükleri**: Kubernetes API üzerinden erişilebilen olay günlükleri, cluster'daki değişiklikleri ve etkinlikleri izlemenize yardımcı olabilir. Bu günlükler, pod'ların başlatılması, güncellenmesi veya silinmesi gibi olayları içerir.

- **Olay Günlüklerini İzleme**: Olay günlüklerini izleyerek, cluster'da gerçekleşen değişiklikleri anında takip edin ve gerekirse müdahale edin.

Sonuç

Kubernetes bileşenleri için günlük yönetimi, cluster'ın sağlığını ve performansını izlemek için kritik öneme sahiptir. Merkezi log toplama, uygun günlük seviyeleri ve analiz araçları kullanmak, operasyonel verimliliği artırmanın yanı sıra hızlı sorun giderme ve yanıt verme yeteneğinizi iyileştirir. Günlük yönetimi, Kubernetes ortamlarında güvenilirlik ve kararlılık sağlamak için temel bir unsurdur.

KAYNAKÇA

CHATGPT

https://gokhana.medium.com/microservice-mimarisi-nedir-microservice-mimarisine-giri%C5%9F-948e30cf65b1

https://ufuk.in/2023/09/08/mikroservis-mimarisi-nedir-avantajlari-nelerdir/

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

https://medium.com/@resittasdemir/monolitik-mimari-ve-mikroservis-mimarisi-33ae65d41a26 (resim 1.1)

https://tarangsharma.hashnode.dev/docker-engine-architecture (RESİM)

https://www.linkedin.com/pulse/what-docker-how-create-image-execute-application-within-varun-lobo (RESİM)

https://www.linkedin.com/pulse/docker-build-ship-run-any-app-anywhere-ramesh-babu-chayapathi (resim)

https://docker-k8s-lab.readthedocs.io/en/latest/docker/docker-etcd.html (resim)