

Kubernetes

video1: https://www.youtube.com/watch?v=8wKpB_xtPgs

Bu video, yazılım geliştirme dünyasında önemli bir yere sahip olan mikroservis mimarisi ve Kubernetes teknolojisini kapsamlı bir şekilde ele almaktadır. Başlangıçta, monolitik mimarının ne olduğu, neden büyüyen projelerde yetersiz kaldığı ve ölçeklenme problemlerine nasıl yol açtığı anlatılmaktadır. Monolitik mimarının temel sorunları, büyük ve karmaşık projelerde bakım, yeniden dağıtım ve ölçeklendirme süreçlerinin zorlaşmasıdır. Bu sorunlar, yazılım dünyasında mikroservis mimarisine geçiş zorunlu kılmıştır. Mikroservis mimarisi, büyük yazılım sistemlerini bağımsız, küçük ve yönetilebilir servislere bölgerek bu problemlere çözüm sunar.

Videoda ayrıca Google'ın, monolitik mimariden mikroservis mimarisine geçiş süreci ve bu alandaki deneyimlerinden doğan Borg adlı iç araştırmalarının bahsedilmektedir. Borg, Google'ın milyonlarca servisini yönetmek için geliştirdiği güçlü bir konteyner orkestrasyon sistemidir. Kubernetes ise Borg'un açık kaynaklı ve daha sadeleştirilmiş bir versiyonu olarak geliştirilmiş ve dünya genelinde yaygınlaşmıştır. Kubernetes, mikroservislerin konteynerler içinde yönetilmesini, ölçeklendirilmesini ve otomatikleştirilmesini sağlar.

Video, Kubernetes'in küçük projelerde gereksiz karmaşıklık yaratabileceğini ve her projede kullanılmasının zorunlu olmadığını vurgulamaktadır. Mikroservis mimarisine geçişin ve Kubernetes kullanımının, ancak projenin büyümesi ve karmaşık hâle gelmesi durumunda anlam kazandığı belirtilmektedir. Ayrıca, Kubernetes'in Google tarafından açık kaynak olarak paylaşılması ve diğer büyük şirketler tarafından benimsenmesi anlatılmaktadır. Son olarak, videoda ilerleyen bölümlerde Kubernetes hakkında daha detaylı içeriklerin sunulacağına dair bilgi verilmektedir.

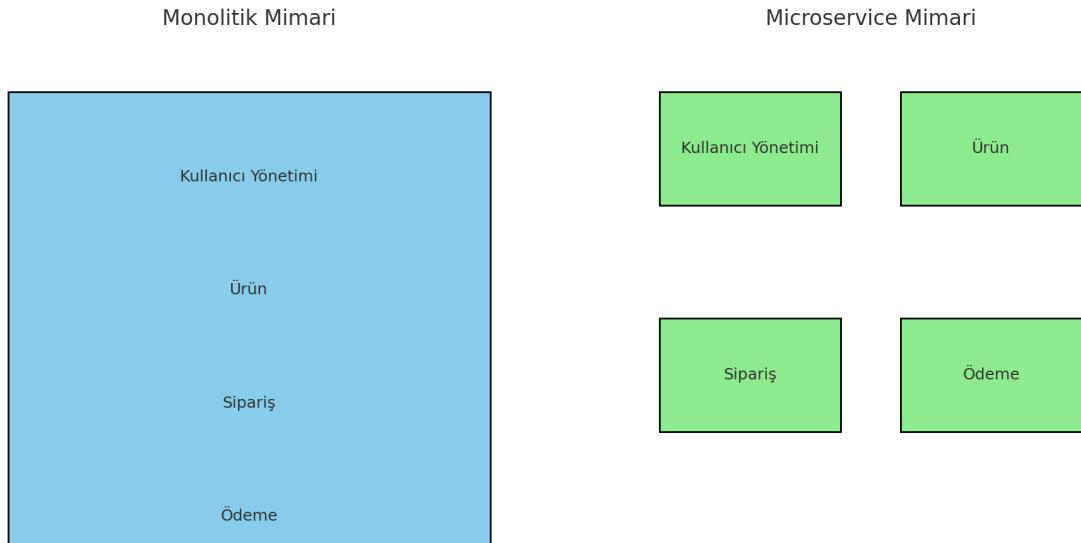
Öne Çıkanlar

- Mikroservis mimarisi, büyük ve karmaşık yazılım projelerini küçük, bağımsız servislere bölgerek yönetilebilir kılar.
- Monolitik mimaride tüm işlemler tek bir proje içinde olduğundan büyündükçe yönetim ve ölçeklendirme zorlukları artar.
- Google, 2000'li yılların başında monolitik yapıdan mikroservise geçiş yaparak büyük sistemlerini daha verimli yönetmeye başladı.
- Borg, Google'in mikroservisleri konteynerler içinde yönetmek ve ölçeklendirmek için geliştirdiği güçlü iç araçtır.
- Kubernetes, Borg'un açık kaynaklı, kullanıcı dostu ve sadeleştirilmiş versiyonu olarak ortaya çıkmıştır.
- Kubernetes, projeler büyündüğünde ve mikroservis mimarisi kullanıldığında anlam kazanır; küçük projelerde gereksiz karmaşıklık yaratabilir.
- Büyük teknoloji şirketleri Kubernetes'i benimseyip kendi bulut ve altyapı çözümlerine entegre etmiştir.

Temel İçgörüler

- **Monolitik Mimarının Sınırlamaları:** Monolitik yapılar başlangıçta basit ve uygulanması kolay olsa da, projeler büyündükçe sürekli yeniden dağıtım (redeploy), bakım zorlukları ve yetersiz ölçeklenme nedeniyle ciddi sorunlar ortaya çıkar. Bu sorunlar, özellikle modülerlik ve bağımsız geliştirme gereksinimleri arttığında daha belirgin hale gelir.
- **Dikey ve Yatay Ölçeklendirme Arasındaki Farklar:** Monolitik mimaride genellikle dikey ölçeklendirme (daha güçlü sunucular kullanmak) tercih edilirken, mikroservis mimarisinde yatay ölçeklendirme (aynı servisin birden fazla kopyasını kullanmak ve yükü dağıtmak) esnekliği sağlar. Bu da kaynakların daha verimli kullanılmasına ve sistemin daha dayanıklı olmasına olanak tanır.
- **Mikroservislerin Yönetim Zorlukları ve Gereksinimleri:** Mikroservis mimarisi, bağımsız servislerin yönetimi için gelişmiş araçlar gerektirir. Servislerin durumunun izlenmesi, ölçeklendirilmesi, hata durumunda yeniden başlatılması, IP adreslerinin dinamik yönetimi gibi karmaşık operasyonlar, Kubernetes gibi orkestrasyon sistemleri olmadan zor ve hataya açık hale gelir.

-  **Google'ın Borg'u ve Kubernetes'in Doğuşu:** Google'ın geliştirdiği Borg sistemi, konteynerler içinde çalışan servislerin yönetimini kolaylaştırmak için tasarlanmış kapsamlı bir orkestrasyon sistemidir. Kubernetes ise Borg'un deneyimlerinden yola çıkarak, açık kaynak olarak geliştirilen, daha kullanıcı dostu ve esnek bir çözüm olarak piyasaya sürülmüştür. Bu sayede Kubernetes, dünya çapında yaygınlaşmıştır.
 -  **Kubernetes'in Ekosistemdeki Rolü:** Kubernetes, sadece konteyner yönetimi değil, aynı zamanda yük dengeleme, otomatik ölçeklendirme, servis keşfi ve hata toleransı gibi birçok kritik işlevi otomatikleştirir. Bu da mikroservis mimarisini benimseyen şirketlerin yüksek ölçeklenebilirlik ve dayanıklılık elde etmesini sağlar.
 -  **Kubernetes'in Her Proje İçin Uygun Olmaması:** Küçük ve basit projelerde Kubernetes kullanımı gereksiz karmaşıklık ve yönetim yükü getirebilir. Bu nedenle Kubernetes, sadece mikroservis mimarisi kullanan ve büyüyen projelerde anlam kazanır. Proje monolitik yapıda ise Kubernetes kullanımı çoğunlukla gereksizdir ve geliştiricilere ek zorluklar çıkarabilir.
 -  **Teknoloji Devlerinin Stratejik Hamlesi:** Google, Kubernetes'i açık kaynak olarak sunarak bulut pazarında rekabeti artırmayı ve kendi liderliğini pekiştirmeyi hedeflemiştir. Amazon, Microsoft gibi diğer büyük oyuncular da Kubernetes'i kendi bulut platformlarına entegre ederek ekosistemi zenginleştirmiştir. Bu durum, Kubernetes'in global teknoloji altyapısında kritik bir yer edinmesini sağlamıştır.
-



İşte **Monolitik mimari** (solda) ile **Microservice mimarisi** (sağda) arasındaki farkı gösteren basit bir çizim .

- Solda: Tüm modüller **tek bir yapı** içinde.
- Sağda: Her özellik **bağımsız bir servis** olarak ayrılmış.

video2: https://www.youtube.com/watch?v=ZE28_IrjRmw

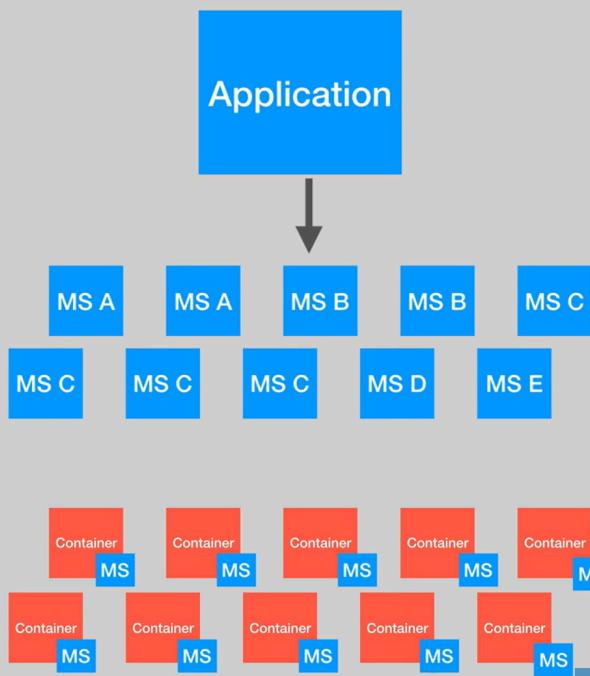
Monolith



Microservices

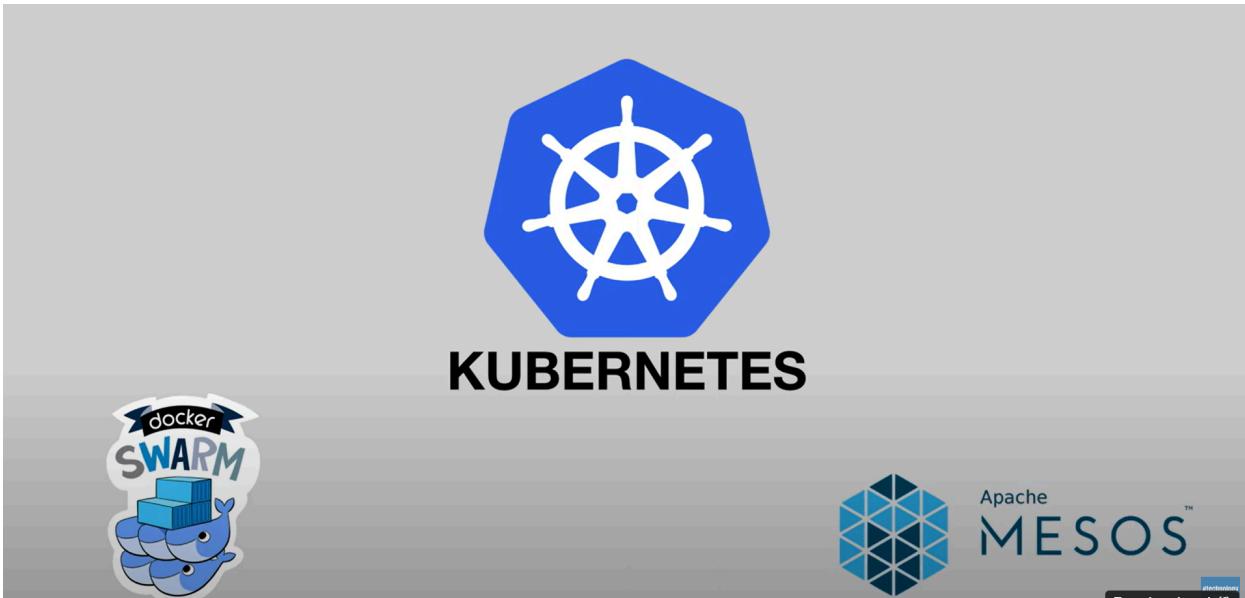


Containers



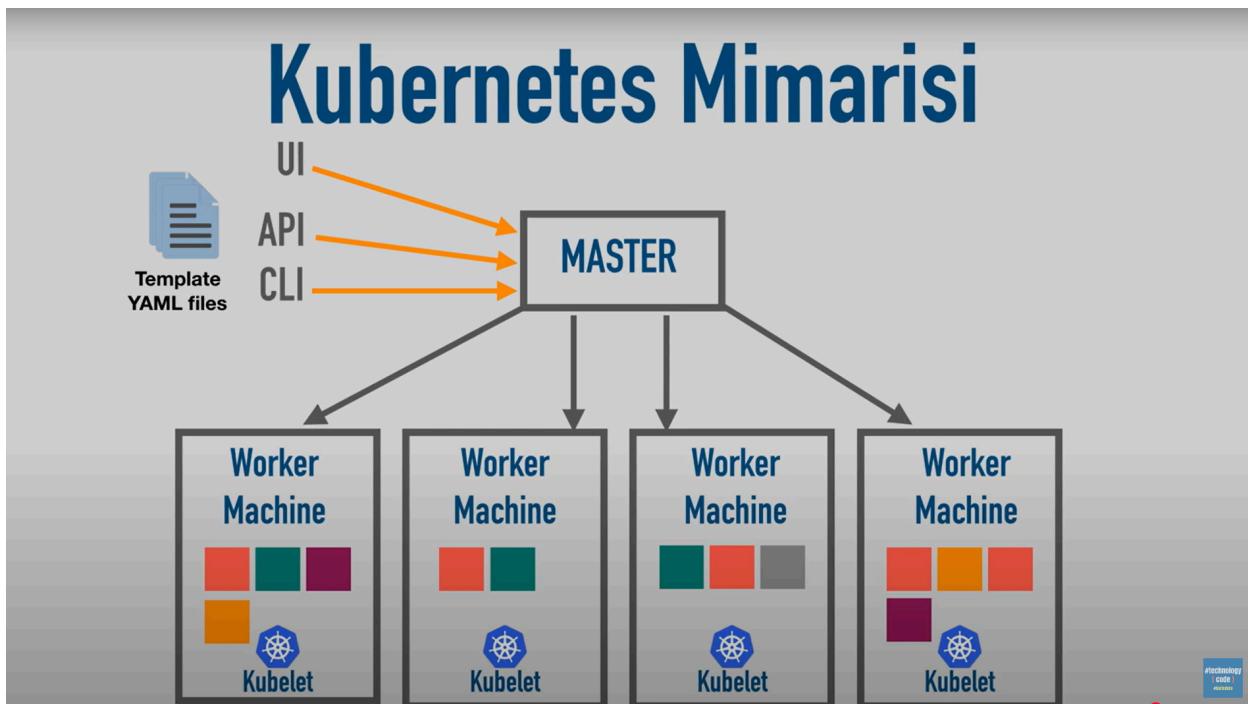
Container Orchestration

- Provisioning and deployment (IT altyapısının hazırlanımı ve dağıtım)
- Configuration and scheduling (Konfigürasyon ve Zamanlama)
- Resource allocation (Kaynak tahsis)
- Container availability (Konteyner kullanılabilirliği)
- Scaling or removing containers based on balancing workloads across your infrastructure (Altyapınızdaki iş yüklerini dengelemeye dayalı olarak workload birimlerini ölçeklendirme veya kaldırma)
- Load balancing and traffic routing (Yük dengeleme ve trafik yönlendirme)
- Monitoring container health (Konteynerlerin çalışma durumlarını izleme)
- Keeping interactions between containers secure (Konteynerler arası etkileşimleri güvende tutmak)



Kubernetes

- Açık kaynak kodlu Container Orchestration aracı
- Google tarafından yazıldı
- Şu an CNCF (Cloud Native Computing Foundation) bünyesi altındadır
- 1000lerce konteynırları aynı anda yönetebilir
- On-Premise - Cloud - Hybrid farketmeksiz



Kubernetes != Docker

Kubernetes, Docker olmadan çalışabilir aynı şekilde Docker da Kubernetes olmadan çalışabilir

Bu videoda Kubernetes'in ne olduğu, ortaya çıkış nedenleri ve hangi problemleri çözdüğü detaylı şekilde anlatılmaktadır. Öncelikle konteyner kavramı açıklanmakta; konteynerlerin, uygulamaların ve bağımlılıklarının standart bir yazılım biriminde paketlenerek farklı ortamlarda hızlı ve güvenilir şekilde çalıştırılmasını sağladığı vurgulanmaktadır. Monolitik yazılım yapısından mikroservis mimarisine geçişle birlikte, mikroservislerin konteynerler halinde paketlenmesi ve yönetilmesinin önem kazandığı belirtilmektedir. Burada Kubernetes gibi konteyner orkestrasyon araçlarının devreye girdiği, bu araçların uygulamaların otomatik ölçeklenmesi, yük

dengelenmesi, hata durumlarında hızlı toparlanma gibi kritik işlevleri üstlendiği anlatılmaktadır.

Video, gerçek yaşam senaryolarıyla, değişken kullanıcı taleplerine göre otomatik ölçeklendirme, sunucu çökmesi durumunda uygulamaların otomatik yeniden başlatılması ve sistemin kesintisiz çalışmasının sağlanması gibi problemlere Kubernetes'in nasıl çözümler getirdiğini göstermektedir. Ayrıca, uygulama dağıtımını (deployment) sırasında yaşanan kesintiler ve versiyon yükseltme (upgrade) süreçlerindeki zorluklar ele alınarak, Kubernetes'in bu süreçlerde otomatik dağıtım ve geri alma (rollback) mekanizmaları sunduğu ifade edilmektedir.

Konteyner orkestrasyonun ne olduğu, sağladığı hizmetler (altyapı hazırlığı, kaynak yönetimi, yük dengeleme, güvenlik, izleme vb.) ve Kubernetes'in bu alandaki en popüler açık kaynak aracı olduğu belirtilmektedir. Kubernetes'in mimarisi, master ve worker node'lar, kubelet ajanının işleyışı ve komutların nasıl iletiliği açıklanmaktadır. Son olarak Docker ve Kubernetes arasındaki farklara değinilmekte; Docker'ın konteyner oluşturma ve çalışma teknolojisi olduğu, Kubernetes'in ise konteynerleri yönetip orkestre eden bir sistem olduğu vurgulanmaktadır.

Öne Çıkan Noktalar

- 🐳 Konteynerler, uygulamaların bağımlılıklarıyla birlikte paketlenmesini sağlayan standart yazılım birimleridir.
- ⚙️ Mikroservis mimarisi, uygulamaların küçük ve bağımsız parçalar halinde yönetilmesini kolaylaştırır.
- 📈 Kubernetes, uygulamaların otomatik ölçeklendirilmesi ve yük dengesinin sağlanması için geliştirilmiştir.
- 🔄 Kubernetes, uygulama dağıtımında kesintisiz geçiş ve hata durumunda otomatik geri alma (rollback) imkanı sunar.
- 💻 Kubernetes mimarisi, bir master ve çoklu worker node'lardan oluşur; komutlar master üzerinden yönetilir.
- 🔒 Konteyner orkestrasyonu, kaynak yönetimi, güvenlik ve izleme gibi kritik operasyonları otomatikleştirir.

-  Docker, konteyner oluşturma teknolojisi iken Kubernetes, bu konteynerlerin yönetimi için kullanılan orkestrasyon aracıdır.

Temel İçgörüler

-  **Konteynerlerin Sağladığı Taşınabilirlik:** Konteynerler, uygulamaların bağımlılıklarıyla birlikte paketlenip farklı ortamlar arasında sorunsuz taşınmasını sağlar. Bu, geliştirme ve üretim ortamları arasındaki uyumsuzlukları ortadan kaldırarak yazılım teslimat süreçlerini hızlandırır ve güvenilir kılar. Özellikle mikroservis mimarisinde bu taşınabilirlik, geliştirme süreçlerini çok daha esnek hale getirir.
-  **Otomatik Ölçeklendirme İhtiyacı:** Kullanıcı trafigindeki dalgaların, sistem kaynaklarının verimli kullanılmasını zorunlu kılar. Kubernetes, gelen trafik miktarına göre uygulamaların sayısını otomatik olarak artırıp azaltarak, hem maliyet etkinliği sağlar hem de yüksek performansı garanti eder. Bu sayede aşırı yüklenme veya kaynak israfı önlenir.
-  **Yüksek Erişilebilirlik ve Hata Toleransı:** Sunucuların veya uygulamaların arızalanması durumunda Kubernetes, otomatik olarak sağlıklı çalışan yeni konteynerler başlatarak sistem kesintilerini minimize eder. Bu özellik, kritik iş yüklerinin sürekli çalışmasını sağlar ve işletmelerin iş sürekliliğini korumasına yardımcı olur.
-  **Kesintisiz Dağıtım ve Rollback:** Uygulama yeni bir sürümü güncellenirken, Kubernetes farklı stratejilerle kesintisiz hizmet sunar. Yeni sürüm sorunsuz çalışmazsa otomatik rollback ile eski sürüm hızlıca dönmek mümkündür. Bu, üretim ortamında hataların etkisini azaltır ve kullanıcı deneyimini korur.
-  **YAML Tabanlı Konfigürasyon Yönetimi:** Kubernetes, sistem yapılandırmasını YAML dosyalarıyla yönetir. Bu yapı, altyapının kod olarak yönetilmesini (Infrastructure as Code) mümkün kılarak, sürümlendirme, tekrarlanabilirlik ve otomasyon avantajları sunar. Bu da DevOps süreçlerinin temel taşlarından biridir.
-  **Master-Worker Mimarisi ve Kubelet:** Kubernetes'in master node'u tüm sistemi yönetirken, worker node'lar uygulamaların çalıştığı fiziksel veya sanal makineler olarak görev yapar. Her worker üzerinde çalışan kubelet ajani,

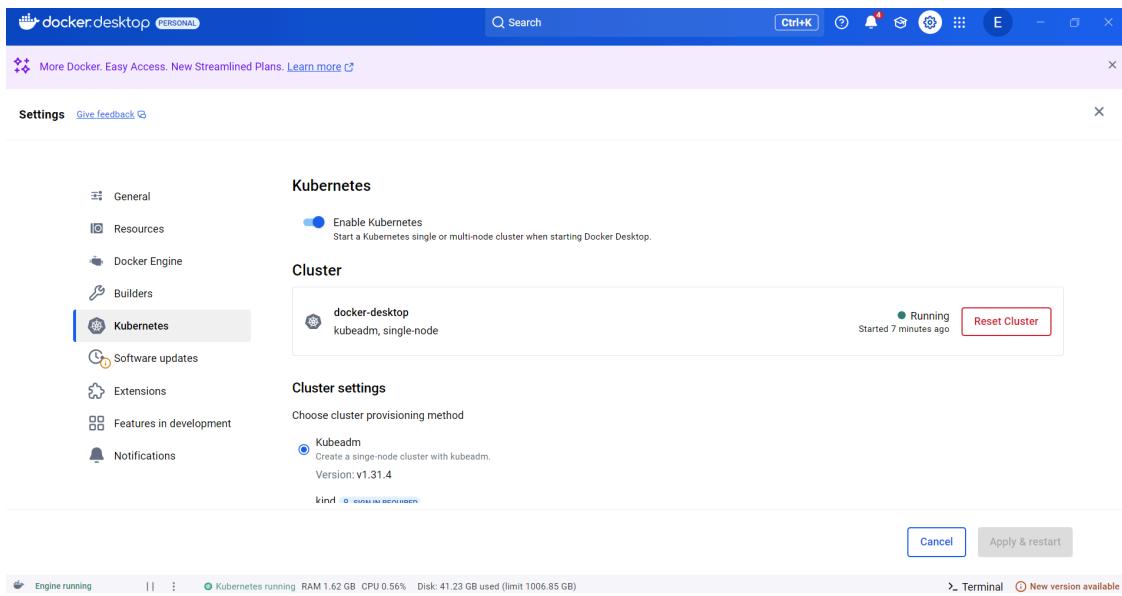
master ile iletişimde olup komutları uygular. Bu mimari, sistemin ölçülebilirliğini ve yönetilebilirliğini artırır.

-  **Docker ve Kubernetes'in Ayrımı:** Docker, konteyner oluşturma ve çalışma teknolojisi olarak temel bir yapıştırıdır. Kubernetes ise bu konteynerleri yönetmek ve orkestre etmek için kullanılan araçtır. Birlikte kullanılabılır ancak birbirlerinin yerine geçmezler. Bu farkın bilinmesi, doğru teknoloji tercihleri yapılması açısından kritik öneme sahiptir.

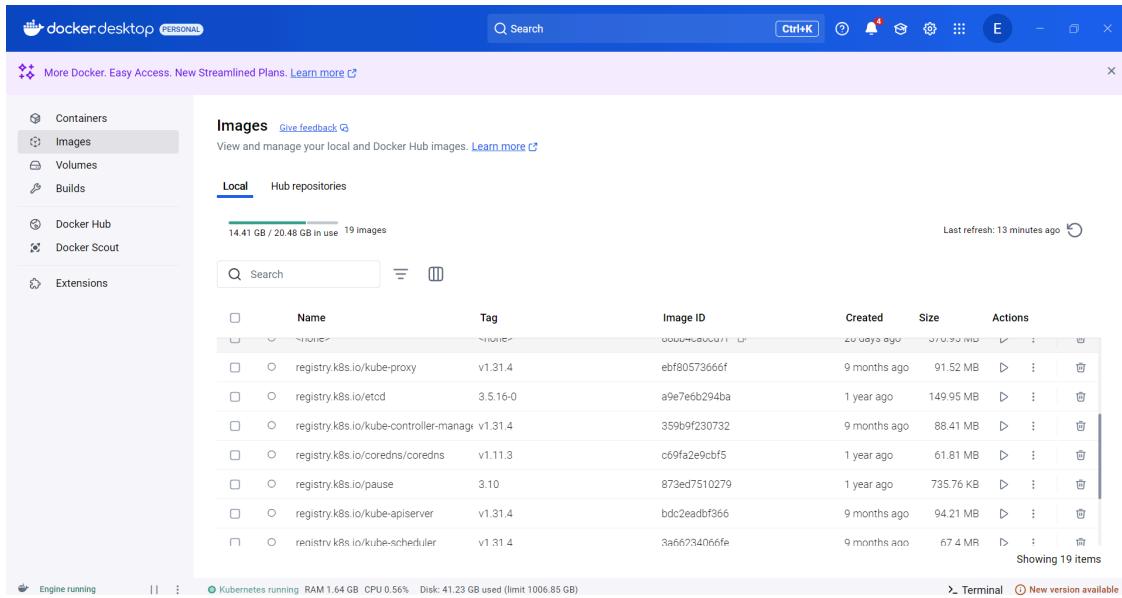
Bu video, Kubernetes'i ve konteyner orkestrasyonunu temel kavramları ve gerçek dünya senaryolarıyla anlaşılır şekilde açıklayarak, teknolojiye yeni başlayanlar için sağlam bir temel oluşturur. Ayrıca, mikroservislerin yönetiminde ve bulut tabanlı sistemlerde yaşanan zorluklara karşı modern çözümler sunan Kubernetes'in neden bu kadar popüler olduğunu açıklar.

video serisi: [https://youtube.com/playlist?
list=PLoOLmxD7O9vD42O8o6I_mUGRDDRjri197&si=7cgkJp9oUHHhauiz](https://youtube.com/playlist?list=PLoOLmxD7O9vD42O8o6I_mUGRDDRjri197&si=7cgkJp9oUHHhauiz)

video serisi 1: [https://www.youtube.com/watch?
v=IJJ6F1grEKM&list=PLggmNmVfeLlbICjiTR1AUAF7BhKxNvQzS&index=1&t=25s](https://www.youtube.com/watch?v=IJJ6F1grEKM&list=PLggmNmVfeLlbICjiTR1AUAF7BhKxNvQzS&index=1&t=25s)



Docker Desktop >> Settings >> Kubernetes



Kubernetes Images

Kubernetes Running

```
kubectl get nodes
```

```
C:\Users\tokel>cd C:\Users\tokel\Desktop  
C:\Users\tokel\Desktop>kubectl get nodes  
NAME           STATUS    ROLES      AGE   VERSION  
docker-desktop   Ready     control-plane   14m   v1.31.4
```

komut çalışan Kubernetes i gösterir

kubectl indir

<https://kubernetes.io/docs/tasks/tools/>

<https://youtu.be/SJQFGPlw04E>

```
C:\Users\tokel\Downloads>kubectl version --client=true  
Client Version: v1.31.0  
Kustomize Version: v5.4.2
```

```
C:\Users\tokel\Downloads>cd ..  
C:\Users\tokel>kubectl version --client=true  
Client Version: v1.31.4  
Kustomize Version: v5.4.2
```

Bu video, Kubernetes'in temel mimarisi, çalışma prensipleri ve ana bileşenleri hakkında kapsamlı bir açıklama sunmaktadır. Kubernetes, konteynerleştirilmiş uygulamaların otomatik yönetimi için kullanılan bir orkestrasyon aracıdır. Video, Kubernetes'in gerçek hayatı bir orkestra şefine benzetilerek, birçok farklı konteynerin uyum içinde ve otomatik olarak nasıl yönetildiği anlatılmaktadır. Ayrıca, Kubernetes'in bileşenleri (API Server, etcd, Controller Manager, Scheduler, CoreDNS, vb.) ve bunların işlevleri detaylandırılmıştır. Kubernetes'in, fiziksel veya sanal nodlar üzerinde çalışan konteynerlerin yaşam döngüsünü kontrol ettiği, hata durumlarında otomatik müdahale ettiği ve dağıtım objeleri (deployment) ile podları yönettiği vurgulanmıştır. Büyük ölçekli sistemlerde binlerce konteynerin ve cluster'ın yönetilmesinin insan gücüyle mümkün olmadığı, bu nedenle otomasyonun zorunlu olduğu anlatılmaktadır. Konteynerlerin podlar içerisinde çalıştığı, deployment objelerinin bu podların yönetiminden sorumlu olduğu, servis objelerinin ise ağ iletişimini sağladığı belirtilmiştir. Son olarak, Kubernetes'in deklaratif yöntemle yönetildiği, verilen emirlerin etcd veri tabanında tutulduğu ve sistemin bu bilgilerle otomatik olarak çalıştığı aktarılmıştır.

Öne Çıkanlar

- 🎻 Kubernetes, konteynerleri organize eden bir orkestra şefi gibi çalışır.
- ☁ Cloud sistemleri Kubernetes veya benzeri orkestrasyon araçları kullanarak uygulamaları yönetir.
- 📦 Kubernetes, konteynerlerin yaşam döngüsünü otomatik olarak yönetir ve hata durumlarında müdahale eder.
- 💻 Nodlar, fiziksel veya sanal makineler olup konteynerlerin çalıştığı altyapayı oluşturur.
- 🔧 Podlar, konteynerlerin çalıştığı en küçük yönetim birimidir.
- 🔄 Deployment objeleri podların yönetimini ve durum takibini sağlar.
- 🌐 Servis objeleri, ağ iletişimini ve uygulamalar arası bağlantıyı yönetir.

Temel İçgörüler

- 🎻 **Kubernetes'in Orkestrasyon Mantığı ve Gerçek Hayat Alegorisı:** Kubernetes, farklı konteynerlerin senkronize ve uyumlu şekilde çalışmasını sağlayan bir orkestrasyon aracıdır. Gerçek hayatı bir orkestra şefi

gibi, Kubernetes de sistemdeki her bir konteynerin görevini koordine eder. Bu benzetme, karmaşık sistemlerin yönetimini basitleştirir ve Kubernetes'in neden gerekli olduğunu açıklar. Bu yaklaşım, özellikle büyük ölçekli uygulamalarda manuel müdahalenin imkânsız olduğu durumlarda otomasyonun önemini vurgular.

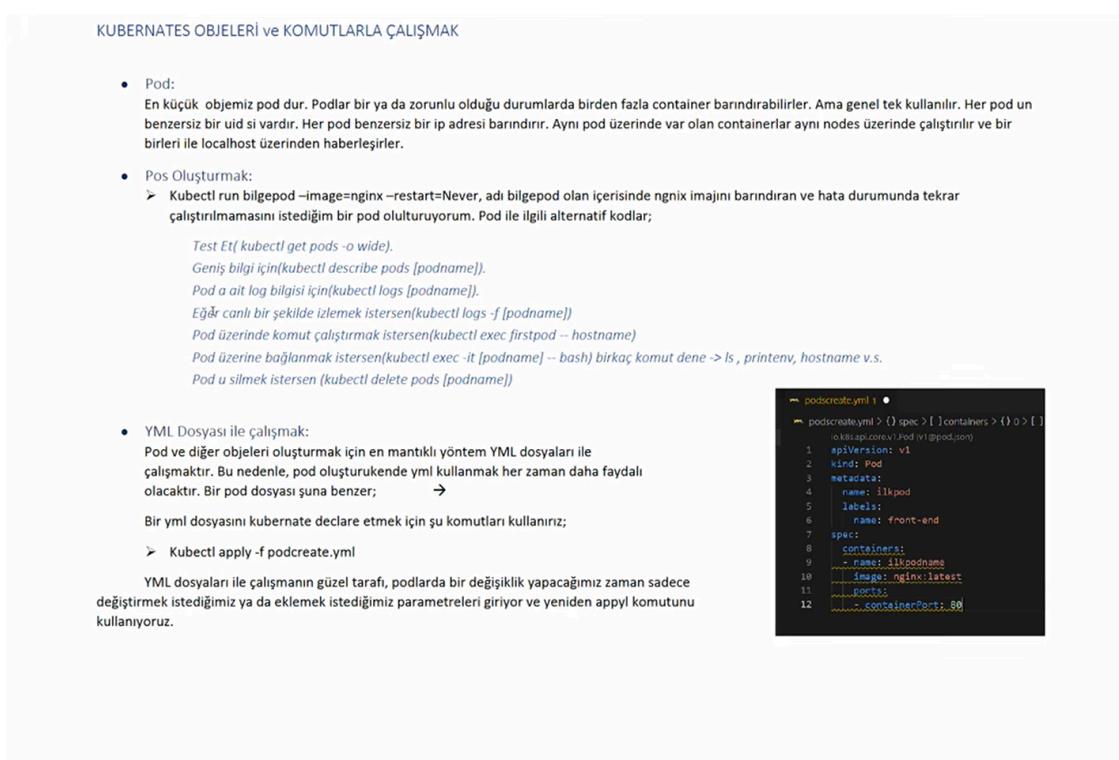
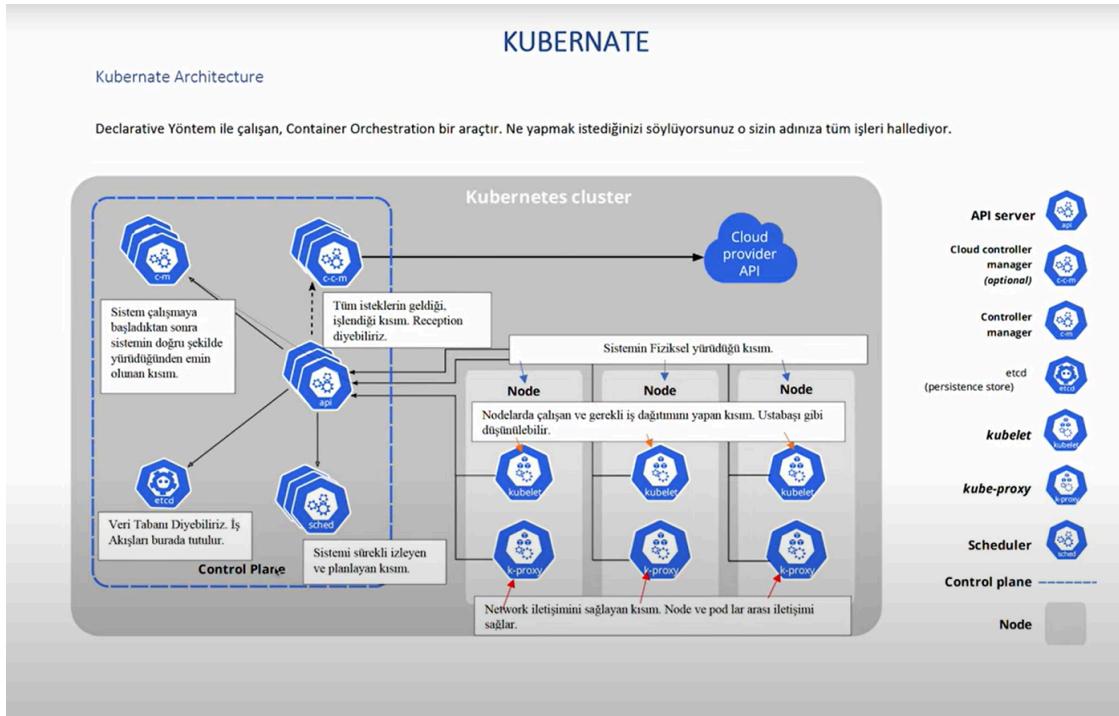
-  **Bulut Sistemleri ve Kubernetes Uyumu:** Google tarafından geliştirilen Kubernetes, günümüzde bulut altyapılarında en yaygın kullanılan orkestrasyon aracıdır. Amazon gibi diğer bulut sağlayıcıları, kendi orkestrasyon çözümlerini geliştirmiş olsa da Kubernetes'in standart hale gelmesi, mikroservis mimarileri ve konteyner bazlı uygulamalar için avantaj sağlar. Bulut sağlayıcılarının kendi çözümlerini teşvik etmeleri, Kubernetes maliyetlerini artırmak ve kendi sistemlerine kullanıcı çekmek için stratejik bir hamledir.
-  **Konteynerlerin Yaşam Döngüsü Yönetimi:** Kubernetes, konteynerlerin otomatik olarak başlatılması, durması ve gerektiğinde yeniden başlatılması işlemlerini yönetir. Bu, uygulamaların kesintisiz çalışması için kritik öneme sahiptir. Özellikle yüksek erişilebilirlik gerektiren sistemlerde, konteynerlerin durumunun sürekli izlenmesi ve anında müdahale edilmesi, sistem güvenilirliğini artırır.
-  **Nodlar ve Sanallaştırma Yaklaşımı:** Nodlar, Kubernetes kümesindeki fiziksel veya sanal makineleri temsil eder ve konteynerlerin çalıştığı altyapıyı oluşturur. Sanallaştırılmış bu makineler, kaynakların verimli kullanılmasını sağlar. Çoklu nodların bir araya gelmesi, ölçeklenebilirlik ve yüksek performans sağlar. Nodların bir fabrika içindeki makineler gibi düşünülmesi, sistemin genel işleyişini anlamayı kolaylaştırır.
-  **Podlar ve Mikroservis Mimarisi:** Podlar, Kubernetes'in en küçük çalışma birimidir ve genellikle tek bir konteyner ya da birden fazla konteyneri bir arada barındırır. Mikroservis mimarisi ile uyumlu olan podlar, uygulamaların modüler ve bağımsız olarak yönetilmesini sağlar. Bu yapı, uygulama bileşenlerinin kolayca güncellenmesi, ölçeklendirilmesi ve hatalara karşı izole edilmesini mümkün kılar.
-  **Deployment ve Yönetimsel Katman:** Deployment objeleri, podların yönetiminden ve yaşam döngüsünün takibinden sorumludur. Podların kaç tane çalışacağını, hangi versiyonun kullanılacağını ve hata durumunda yeniden başlatma işlemlerini deklaratif olarak belirler. Bu sayede, sistem yöneticileri ve

geliştiriciler manuel müdahaleye gerek kalmadan uygulamaların durumunu kontrol edebilir ve sürdürülebilir bir operasyon sağlar.

-  **Servisler ve Ağ Yönetimi:** Kubernetes servis objeleri, podlar arasındaki iletişimini ve dış dünyaya açılan bağlantıları yönetir. Servisler, yük dengeleme, keşif ve ağ trafiğinin doğru yönlendirilmesi gibi kritik işlevleri üstlenir. Bu, büyük ve dağıtık mimarilerde uygulamaların birbirleriyle sağlıklı iletişim kurmasını garanti eder.
-  **Deklaratif Yönetim ve Otomasyon:** Kubernetes, sistemin istenilen durumunu deklaratif olarak belirtmeye dayanır. Kullanıcılar, sistemin hangi durumda olması gerektiğini belirtir; Kubernetes ise bu durumu sağlamak için gerekli adımları otomatik olarak atar. Bu yaklaşım, hem yönetim kolaylığı sağlar hem de insan hatalarını minimize ederek sistem kararlılığını artırır. Etcd veri tabanı, bu istenilen durumun saklandığı merkezi bir konumdur ve tüm Kubernetes bileşenleri buradaki bilgiyi kullanarak koordineli çalışır.
-  **Hata Yönetimi ve Otomatik Müdahale:** Kubernetes'in en önemli avantajlarından biri, nod ya da pod kapanması durumunda otomatik olarak yeni podların devreye alınmasıdır. Bu, üretim ortamlarında kesintisiz hizmet sağlamak için kritik bir özelliklektir. Sistem, gerçek zamanlı olarak kaynakları izler ve gerektiğinde müdahale ederek uygulamanın çalışmaya devam etmesini sağlar.
-  **Büyük Ölçekli Sistemlerde Yönetim Zorlukları:** Video, Trendyol örneğiyle binlerce nod ve yüz binlerce pod'un nasıl yönetildiğini göstererek Kubernetes'in neden zorunlu olduğunu netleştirir. İnsan gücüyle bu kadar büyük bir altyapıyı yönetmek imkânsızdır. Bu da Kubernetes gibi otomasyon araçlarına olan ihtiyacı gösterir.

Bu kapsamlı açıklamalar, Kubernetes'in karmaşık sistemlerin yönetiminde nasıl merkezi bir rol oynadığını, hangi bileşenlerle çalıştığını ve modern bulut tabanlı uygulama geliştirme süreçlerinde neden vazgeçilmez olduğunu derinlemesine anlamamıza olanak tanır.

video serisi 2: <https://youtu.be/Wlj94El4g5Q?si=vWLcMMmItHalYABf>



- Deployment:
Burada deployment objesinin kavramak için senaryolarımızı değerlendirelim.
- Yml dosyası ile bir pod oluşturduk, podumuz A numaralı node üzerinde çalışıyor diyelim. Bir sorun oldu ve node kapandı. Böyle bir durumda kube-sched pod u başka bir node üzerinde çalışmaz arkadaşlar bu nedenle projemiz down olur.
 - Bu sorunu aşmak için her node üzerinde çalışmak üzere örneğin, 10 pod.yml dosyası yaptık ve tüm node lar üzerinde çalıştırıldık diyelim. Peki uygulamamızı güncellememiz gerekirse ne olacak? Yml dosyasını değiştireceğim, sonra tüm node larda podları tekrar güncellemem gerekecek, işler tekrar karışmaya başlıyor.

İşte tam burada Deployment, bizim adımıza istenilen durum ile mevcut durumu kontrol etmek üzere çalışır. Eğer sisteme bir sorun var ise pod umuzu tekrar ayağa kaldırır. Gerekirse bir node tan diğer bir node ta ayağa kaldırır. Ama sistemin çalışmasını garanti eder.

Gerek yok ama elle olutmak isterseniz;

- o Kubectl create deployment ilkdeploy --image=nginx:latest --replicas=2
- o Kubectl scale deployment ilkdeploy --replicas=5

Burada kontrol yapmak için bir bash içinde -w ile izleme ye geçin, diğer bash üzerinde "delete pods [podname]" ile herhangi bir pod u silip sistemi takip edin.

YML ile çalışmak -> Burada önemli olan selector kavramı hangi deployment in hangi podları yöneteceğin burada belirlenir.

Önemli Not: RS(ReplicaSet), Deployment çoklu pod oluştururken önce rs objesi yaratır. Eğer değişim olursa bir rs daha yaratarak yönetimi kararlı hale getirir.

Test etmek için:

- Kubectl get deployment -w
- Kubectl get replicaset -w
- Kubectl get pods -o wide
- Watch kubectl get pods – Linux ta
- Watch kubectl get replicaset – Linux ta
- Değişiklikleri GeriAl -> kubectl rollout undo deployment authdeployment

```
 1  io.k8s.api.apps.v1.Deployment(v1@deployment.json)
 2  apiVersion: apps/v1
 3  kind: Deployment
 4  metadata:
 5    name: authdeployment
 6    labels:
 7      | team: developers
 8  spec:
 9    replicas: 3
10    selector:
11      | matchLabels:
12        | app: backend
13    template:
14      metadata:
15        labels:
16          app: backend
17        spec:
18          containers:
19            - name: nginx
20              image: nginx:latest
21              ports:
22                - containerPort: 80
```

Kubernetes'in temel amacı, uygulamaların konteynerler halinde çalıştığı bir ortamda, bu konteynerlerin yaşam döngüsünü, ölçeklenmesini ve hatalara karşı dayanıklılığını otomatik olarak yönetmektir. Bu özet, verilen transkript üzerinden kubernetes objelerini, mimariyi ve işleyiş tarzını Türkçe olarak sade ve akıcı bir dille açıklamayı amaçlar. Aşağıdaki bölümler, kavramları adım adım ele alarak, gerçek hayattan bir alegoriyle destekleyerek ve teknik terimleri özetleyen bir yapı sunar.

Merkezi Bölüm (Center)

- Kubernetes nedir ve ne yapar?**

Kubernetes, “orchestration tool” olarak, farklı mikroservislerin ve konteynerlerin uyum içinde çalışmasını sağlayan merkezi bir yönetim mekanizmasıdır. Düşünün ki sahnede farklı enstrümanlar var; hepsi aynı anda çalışmalı, ama bir şefin yönetimi olmadan uyum sağlanamaz. Kubernetes bu şef göreviyle çalışır: bileşenleri organize eder, beklenen durumları deklaratif emirlerle tanımlar ve gerektiğinde otomatik müdahaleler yapar.

- Gerçek hayattan alegoriyle açıklama**

- Bir orkestra şefi, müzisyenleri senkronize ederek notaların aynı anda çalmasını sağlar.
 - Kubernetes de bu rolü üstlenir: birden çok konteyneri (mikroservisleri) aynı anda, beklenen durumda çalışır konuma getirir, hatalar olduğunda devreye girer ve otomatik olarak yeniden başlatır.
 - Büyük ölçekli ortamlarda tek başına bir kişinin kontrol edemeyeceği milyonlarca konteyner olduğunda, otomasyon ve deklaratif yapı hayatı önem kazanır. Bu nedenle cloud sağlayıcıları (Amazon, Google gibi) Kubernetes'i kendi altyapılarıyla entegre ederek maliyetleri düşürüp verimliliği artırmayı hedefler.
- **Kubernetes mimarisinin ana parçaları**
 - **Node (Nodelar):** Her biri sanal veya fiziksel bir makineyi temsil eder. Node'lar, üzerinde konteynerlerin (podların) çalışmasını sağlayan fiziksel/virtual güç kaynağıdır.
 - **Podlar:** Konteynerleri içeren en küçük dağıtım birimidir. Her pod, belirli bir uygulama veya veri tabanını içeren bir ya da birkaç konteyneri barındırabilir ve kendi bağımsız çalışma alanını sağlar.
 - **Deployment (Dağıtım):** Pod'ların yaşam döngüsünü yöneten ve istenen durumda kalmalarını sağlayan yönetim objesidir. Pod sayısının artması veya azalması gerekiğinde otomatik olarak ölçeklendirme yapar ve hatalı pod'ları yeniden başlatır.
 - **Service (Servis):** Ağ düzeyinde pod'lar arasında iletişimini sağlayan ve istemcilerin gerekiğinde doğru pod'a yönlendirilmesini garanti eden ağ objesidir.
 - **Etcđ (Güçlü Dağıtık Veri Deposu):** API istekleriyle ilgili konfigürasyonları ve durum bilgilerini güvenli bir şekilde saklar; tüm kontrol düzeyi bu veriler üzerinden karar verir.
 - **CB ctl ve CB shader:** Kontrol paneli üzerinden gelen emirleri alır ve sistemi yönetir. CB shader ise fiziksel ortamı izler ve bir problemi tespit ederse müdahale eder.
 - **Kubernetes bileşenleri ve akış mantığı**

1. Operasyonel bölüm, fiziksel-ya da sanal makineler üzerinde konteynerlerin çalıştığı alanı kapsar.
 2. Yönetim bölümünde ise bu kaynakların ne şekilde ölçüleneceğini, hangi kaynakların hangi anda çalışacağını belirleyen emirler ve politikalar bulunur.
 3. Kontrol düzeyi, API üzerinden gelen istekleri işler, istenen durum ile mevcut durum arasındaki farkı hesaplar ve eksik olanı tamamlar.
 4. State depolama olarak Etcd kullanılır; tüm kararlar ve durumlar bu depoda tutulur ve gerektiğinde geri çağrılır.
- **Geniş ölçekli örnekler ve zorluklar**
 - Bir Trendyol gibi dev e-ticaret servislerinde yüzlerce bölge ve binlerce sanallaştırma katmanı bulunabilir. Burada elle yönetim mümkün değildir; otomasyon, ölçülebilirlik ve dayanıklılık için kritik rol oynar.
 - 270.000 üzerinde çalışan pod, 14 farklı bölgede, 43.000 farklı sanallaştırma ve 3.000 kümelenmiş cluster ile yönetilirken, hangi pod'un çöktüğünü, hangi sürümün çalıştığını ve hangi kaynakların tükenmek üzere olduğunu otomatik olarak izlemek gereklidir.
 - **Kubernetes'in temel objelerinin işlevleri (kısa özet)**
 - Pod: Docker imajını çalıştıran en küçük birim; kendi kaynaklarını ve portlarını tanımlar.
 - Deployment: Pod'ların yaşam döngüsünü kontrol eden ve istenen sayıda kopyayı çalışır durumda tutan yönetim objesi.
 - Service: Pod'lar arası iletişimini ve harici erişimi sağlayan ağ katmanı.
 - Node: Konteynerlerin koşturulduğu fiziksel/üretken makineler. Bu makineler, ihtiyaç halinde ölçeklendirilir.
 - Etcd: Konfigürasyon ve durum bilgisini saklayan güvenli depodur.
 - CB ctl: Kontrol panelinin komutlarını alan arayüz; yeni nod, pod veya diğer objelerin oluşturulmasını sağlar.
 - CB shader: Fiziksel altyapıyı izler ve tespit edilen sorunlarda müdahale eder.

- **Açıklayıcı bir tabloyla kavramlar**

Kavram	Görev	Özet
Node	Fiziksel/virtual makine	Konteynerlerin çalıştığı altyapı gücü sağlar
Pod	Kontejner kümesi	En küçük çalışabilir birim; kendi kaynağı ve portları vardır
Deployment	Yaşam döngü yönetimi	Pod sayısını ve sürümünü denetler; otomatik yeniden başlatır
Service	Ağ iletişimini	Pod'lar arası iletişimini ve erişimi yönetir
Etcđ	Dağıtık veri deposu	Durum ve konfigürasyonu saklar; kararlar buradan alınır
CB ctl	Kontrol arayüzü	Kümeyi programlar; emirler verir
CB shader	İzleme ve müdahale	Fiziksel altyapıyı izler; sorun olduğunda müdahale eder

- **Deklaratif yaklaşımın önemi**

Kubernetes, kullanıcıların temel komutları ve istenen durumları deklaratif olarak ifade etmelerini sağlar. Bu şekilde, "ne yapılacağını" belirtir ve sistem bu hedefe ulaşmak için gerekli adımları otomatik olarak uygular. Böylece operatörlerin her bir karar için manuel olarak müdahale etmesi gerekmek.

- **Kullanıcı için günlük faydalar**

- Kolay ölçeklendirme: talebe göre pod sayısı dinamik olarak artırılır veya azaltılır.
- Yük dengeleme: Servisler aracılığıyla trafiğin dengeli dağılımı sağlanır.
- Yüksek erişilebilirlik: Podlar otomatik olarak yeniden başlatılır, farklı bölgelerde çoğaltılır.
- Otomatik iyileştirme: Arızalı pod gönüllü olarak yeniden başlar veya yer değiştirir; müdahale edilmeden çözümler bulunur.

Sonuç (Outro)

- **Özetle:** Kubernetes, konteyner tabanlı uygulamaları yönetmek için tasarlanmış güçlü bir orkestrasyon aracıdır. Node'lar, pod'lar, deployment'lar ve servisler gibi temel objeler aracılığıyla, büyük ölçekli ve dinamikkökenli ortamlarda bile otomatik içerik yönetimi, hata toleransı ve ölçeklenebilirlik sağlar.
- **İleriye dönük düşünce:** modern bulut mimarilerinde Kubernetes ile otomasyonun ve deklaratif politikanın giderek merkezi hale gelmesi beklenir. Altyapıların daha verimli, güvenli ve dayanıklı hale gelmesi için bu mimari temel bir yapı taşı oluşturmaya devam edecektir.

İşin özü: Kubernetes ile bir orkestra şefi gibi, tüm mikroservisler, kendi zamanında ve senkronize bir şekilde çalar; siz ise görünümdeki akışın belirli akışını tanımlarsınız, gerisi sistem tarafından otomatik olarak yürütülür. Bu nedenle, ölçeklenebilirlik, güvenilirlik ve verimlilik açısından kubernetes, günümüz mikroservis mimarilerinin merkezinde yer alır.

video serisi video 3: <https://www.youtube.com/watch?v=aSw3ZSCmVV0&list=PLggmNmVfeLlbICjiTR1AUAF7BhKxNvQzS&index=3>

• Service
LoadBalancer mantığı için kullanılan bir yapıdır. Sorun farklı nodlarda yaratılan ve farklı bloklarda bulunan podları haberleşmesi için kullanılır. Ayrıca yatay genişlemenin mümkün olmasını sağlar.

Types:
ClusterIP, localde çalışır. (Cluster-ip tabanlıdır) NodePort, localde çalışır.(Cluster-ip tabanlıdır) LoadBalancer, Cloudta çalışır.

```
io.k8s.api.core.v1.Service (v1@service.json)
1 kind: Service
2 apiVersion: v1
3 metadata:
4   name: authloadbalancer
5 spec:
6   selector:
7     | app: backend
8   type: ClusterIP
9   ports:
10    - name: backend
11      port: 8091
12      targetPort: 8091
```

```
io.k8s.api.core.v1.Service (v1@service.json)
1 kind: Service
2 apiVersion: v1
3 metadata:
4   name: authloadbalancer
5 spec:
6   selector:
7     | app: backend
8   type: NodePort
9   ports:
10    - name: backend
11      port: 8091
12      targetPort: 8091
```

```
io.k8s.api.core.v1.Service (v1@service.json)
1 kind: Service
2 apiVersion: v1
3 metadata:
4   name: authloadbalancer
5 spec:
6   selector:
7     | app: backend
8   type: LoadBalancer
9   ports:
10    - name: backend
11      port: 8091
12      targetPort: 8091
```

Burada Service test etmek için;
Herhangi bir pod'a bağlanıp, servis'e ait ip adresi üzerinde istek atabilirsiniz.
nslookup backend ile erişim sağlayabiliyor olmalısınız.
curl backend:8091
➤ Kubectl get endpoints
➤ Kubectl describe endpoints frontend

frontend-deployment

NodePort: Dijital Aşağımız podlarını önde koymalıyız

ClusterIP: İste Kullandığımız podları önde koymalıyız

backend-ip-list
1- 10.0.10.206
2- 10.0.10.204
3- 10.0.10.202
4- 10.0.10.203
5- 10.0.10.205
nev podu ips
7- 10.0.10.206
8- 10.0.10.207

- Liveness probes

Sorun: pod larda sorun olmaya bilir, imajımız indirilmiş olabilir. Ancak uygulamamız çalışmıyor ise ne olacak? İşte burada, bizim servislerinizin ayakta olup olmadığını kontrol eden bir mekanizmadır.

Bizim uygulamalarımız resp-api olduğu için end pointlerimiz sürekli açık olması gereklidir. Bu nedenle bir end point istek atarız ve bu seviyde ayakta ise sorun yoktur. Ancak isteme bir sıkıntı var ise pod un tekrar restart edilmesi gereklidir. İşte bunu sağlamak için liveness probe kullanılır. [httpGet ile](#)

Diyelim ki yaka olması gereken bir DB niz var. Bunu kontrol etmek için belli bir portu dinlemeniz gereklidir bunun için `tcpSocket` kullanırsınız.

- Resources

Kaynakları sınırlı kullanmak için ihtiyacımızı giderir.

```
16      initialDelaySeconds: 15
17      periodSeconds: 20
18      resources:
19          limits:
20              memory: "128Mi"
21              cpu: "500m"
22  ---
```

- Environment Variable

İşletim sistemi, varible tanımlamak için kullanılır. Eğer env görmek istersen

➤ `Kubectl exec [podname] -- printenv`

```
  name: myapp
7   spec:
8     containers:
9       - name: myapp
10      image: <Image>
11      resources:
12        limits:
13          memory: "128Mi"
14          cpu: "500m"
15      ports:
16        - containerPort: 8080
17      env:
```

- Environment Variable

İşletim sistemi, variable tanımlamak için kullanılır. Eğer env görmek isterse

➤ Kubectl exec [podname] -- printenv

```

6   name: myapp
7 spec:
8   containers:
9     - name: myapp
10    image: <Image>
11    resources:
12      limits:
13        memory: "128Mi"
14        cpu: "500m"
15    ports:
16      - containerPort: 8080
17    env:
18      - name: MONGOURL
19      | value: "192.168.1.25"
20      - name: MONGOPORT
21      | value: "29851"
22

```

- Secret

Kubernetes, güvenliğini sağlayamamız gereken bilgileri saklamamız için secret objesini kullanımıza sunar. Prolalar, OAuth token, ssh gibi bilgilerimizi depolar. Gizli bilgileri secret içinde saklamak pod tanımı içinde kullanmaktan daha güvenlidir.

Önemli NOT: oluşturacağımız secret ile atayacağımız pod lar aynı namespace içinde olmalıdır.

Secretleri görmek için;

➤ Kubectl get secrets
➤ Kubectl describe secrets mytoolssecret

```

+ podcreate.yml > {} spec
io.k8s.api.core.v1.Secret(v1@secret.json) v1@secret+v1
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: mytoolssecret
5    type: Opaque
6  stringData:
7    db_mongourl: "192.1.1.2"
8    db_username: "admin"
9    db_password: "S1fffffreeeee"
10 ...
11  apiVersion: v1
12  kind: Pod
13  metadata:
14    name: myapp
15  labels:
16    name: myapp
17  spec:
18    containers:
19      - name: myapp
20        image: <Image>
21        resources:
22          limits:
23            memory: "128Mi"
24            cpu: "500m"
25        ports:
26          - containerPort: 80
27        env:
28          - name: MONGOURL
29          | valueFrom:
30          |   secretKeyRef:
31          |     name: mytoolssecret
32          |     key: db_mongourl
33

```

```

io.k8s.api.core.v1.Secret(v1@secret.json) | io.k8s.ap
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: mytoolssecret
5    type: Opaque
6  stringData:
7    db_mongourl: "192.1.1.2"
8    db_username: "admin"
9    db_password: "S1fffffreeeee"
10 ...

```

- ConfigMap:

Secret ile bire bir aynı işi yapan objedir. Bu obje key-value şeklinde değerleri tutar. Şimdi akılda deli sorular, peki secret ile aynı işi yapıyorsa buna ne gerekvar? Aslında Secret verilerimizi BaseEncode şeklinde tutar ve ayar yaparsak etcd üzerinde auth edilerek kullanılır. Bizim gizli olmayan temel verilerimizi tutacak; Encode, Encrypt edilmeyecek; rahatlıkla erişebileceğimiz bir yapıyı configmap sunar.

```

10  apiVersion: v1
11  kind: Pod
12  metadata:
13    name: configmappod
14  spec:
15    containers:
16      - name: configmappod
17        image: muhammedal155/xxxx
18        resources:
19          limits:
20            memory: "128Mi"
21            cpu: "500m"
22        env:
23          - name: MONGO_URL_POD
24            valueFrom:
25              configMapKeyRef:
26                name: bilgevonfigmap
27                key: MONGO_URL
28          - name: MONGO_ADMIN_POD
29            valueFrom:
30              configMapKeyRef:
31                name: bilgevonfigmap
32                key: MONGO_ADMIN
33        ports:
34          - containerPort: 80

```

```

1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: bilgevonfigmap
5  data:
6    MONGO_URL: "localhost"
7    MONGO_ADMIN: "root"
8    MONGO_PASSWORD: "root"
9  ---

```

- Volume

Senaryo: Diyelim ki bir servisler bütünüümüz var ve bunlar çalışıkları sürece log kayıtlarını tutuyorlar ve üzerinde işlem yaptıkları resim dosyalarını işleyip depoluyorlar. Böyle bir durumda, eğer bizim aktif çalışan pod umuz bir nedenle sorun yaşarsa container yeniden başlatılır yanı mevcut pod silinir yerine yenişi create edilir. Böyle bir durumda depolanan tüm veriler silinir. İşte buna çözüm bulmak için volüme aktif olarak kullanılır.

I

- *emptyDir: volume ilk olarak bir pod oluşturulup bir node atandığında oluşturulur. Ve bu pod o node da çalıştığı sürece var olur. Bu birim başlangıçta boş olarak gelir. Aynı pod üzerinde var olan tüm container lar bu dosya okuma yazma yapabilirler. Eğer pod bir nedenle silinirse tüm veriler gider.(Çok tercih etmem)*
- *hostPath: worker node ta var olan podlara dosya yada dizin bağlama imkanı tanır.(Bunu Hiç Tercih Etmem)*

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: volumepath
5    labels:
6      name: volumepath
7  spec:
8    containers:
9      - name: volumicontainer
10        image: muhammedal155/tinyimagevolume
11        volumeMounts:
12          - name: directory-volume
13            mountPath: /logs
14          - name: file-volume
15            mountPath: /logs/alllogs.log
16        resources:
17          limits:
18            memory: "128Mi"
19            cpu: "500m"
20        volumes:
21          - name: directory-volume # Zaten container üzerinde var olan bir dizini belirtir.
22            hostPath:
23              path: /logs
24              type: Directory
25          - name: directoryOrCreate volume # bir dizin oluştur anıksa eğer bu dizin zaten container üzerinde var ise oluşturma
26            hostPath:
27              path: /temp
28              type: DirectoryOrCreate
29          - name: file-volume # Eğer alllogs.log dosyası yok ise oluşturuyoruz.
30            hostPath:
31              path: /logs/alllogs.log
32              type: FileOrCreate
33

```

ÖNEMLİ: Volume bizim için önemli ancak şuan anlatıklarım bizim ihtiyaçlarımıza karşılık gelmemekte. Bize daha çok bağımsız çalışan volume ler gerekli olmaları ayrıca işleyeceğiz.

Node lara etiket atayarak oluşturacağınız podların etiketli node lar üzerinde çalışmasını sağlayabilirsiniz.

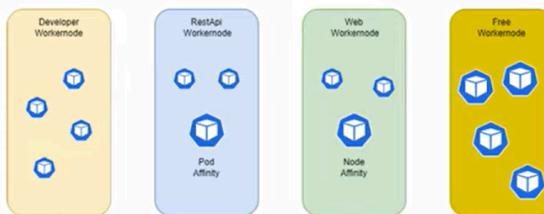
- Pod Affinity:

Podları çalışacağı nod gruplarını belirleyebiliriz.

NOT: Yukarıda kısaca geçtiğim yöntemleri çok karmaşık ve uğraştırıcı olduğu için pek tercih etmiyorum. Zaten her zaman bu kullanımlar isteklerimize yanıt vermiyor. Bu nedenle örneklerini ilgili dosyamiza ekledim.

- Taint ve Toleration:

Sorunu Tanımlayalım; elimizde 3 ve daha fazla workernode olsun(developernode – restapinode – webnode - freenode), gerekli tanımları yaparak rest api olanlar ilgili node ta, web olanlar ilgili node ta ayağa kaldırılsın. Herhangi bir etikete sahip olmayan(nodeaffinity ya da podaffinity olmayan) podları rast gelen nodlar üzerinde ayağa kaldırılacak ve oluşturmak istediğimiz yapı kurgulanamayacaktır.



İşte tam burada, bize gerekli olan node lar üzerinde koşturulacak podların node tarafından belirlenmesi ve koşullarına uymanın pod yapılarını reddetmemesidir. Bu sağlamak için taint ve toleration kullanılır.

Öncelikle sistemimizde çalışan workernode lara bir göz atalım

➢ Kubectl describe nodes minikube ya da docker-desktop

Peki bir node üzerine nasıl ekleme yapılabilir?

➢ Kubectl taint node minikube ya da docker-desktop developerteam=bilgeadam:NoSchedule

➢ Kubectl taint node minikube ya da docker-desktop developerteam- TAIN SILME(sonuna – işaret konulur.)

DİKKAT: burada yazdığım minikube ya da docker-desktop ifadeleri bizim kullandığımız context için. Eğer butta bir node tanımı var ise bu işlemi o node isimleri kullanarak yapmalısınız.

- Job:

Sistemde podların oluşturulması ile ilgili işlemleri deployment lar ile yapabiliriz. Burada sıkıntı şu eğer bizim uygulamamız bir sorunla karşılaştı ve çakıldı. Bu durumda deployment objesi sorunu algılayıp tekrar pod u ayağa kaldırır. Ancak bizim bazen işlemlerini yapıp kapanması gereken podlarımız olabilir, bu durumda sıkıntı çıkacaktır. İşte burada job lar bu işi bizim için yaparlar.

```
└─ _Job.yml > {} spec
    io.k8s.api.batch.v1.Job (v1@job.json)
    1  apiVersion: batch/v1
    2  kind: Job
    3  metadata:
    4    | name: myjob
    5  spec:
    6    ttlSecondsAfterFinished: 100 # eğer job 100sn içinde tamamlanamaz ise sorun var fail et
    7    parallelism: 2 # aynı anda kaçar kaçar pod oluşturulacak
    8    completions: 10 # Kaçtan başarılı job pod oluşturulacak
    9    backoffLimit: 5 # toplam kaç hata olursa işlemleri bırak ve job u kapat 5 kere dene
   10   template:
   11     spec:
   12       containers:
   13         - name: pi
   14           image: perl
   15           command: ["perl", " -Mbignum=bpi", "-wle", "print bpi(2000)"]
   16           restartPolicy: Never #OnFailure
   17
```

Sistemi izlemek için

➢ Kubectl get pods -w

➢ Kubectl get jobs -w

EVEEEET peki job işlemlerinin belli bir zaman diliminde çalışmasını istese idik ne yapackatik?

- CronJob

İşlerin belli bir zaman diliminde yapılmasını sağlamak için kullanılır.

```

1  apiVersion: batch/v1
2  kind: CronJob
3  metadata:
4    name: hello
5  spec:
6    schedule: "* * * * *"
7    jobTemplate:
8      spec:
9        template:
10          spec:
11            containers:
12              - name: hello
13                image: busybox
14                imagePullPolicy: IfNotPresent
15                command:
16                  - /bin/sh
17                  - -c
18                  - date; echo Hello from the Kubernetes cluster
19                restartPolicy: OnFailure
20
21  # | | | | ----- Haftanın günleri (0 - 6)
22  # | | | | ----- Ay (1 - 12)
23  # | | | | ----- Ay ın günü (1 - 31)
24  # | | | | ----- Saat (0 - 23)
25  # | | | | ----- Dakika (0 - 59)
26  # https://crontab.guru/ adresinsen oluşturmak istediğiniz zamanı
27  # deneye bilirsiniz.
28
29  # Örn: 1 * * * * -> 1 dakika sonra
30  # Örn: 1 * 5 * * -> Ayın 5. günü 1. dakika da 2022-03-05 00:01:00
31
32  # Örn: */1 * * * * -> Her 1 dakika da bir çalış
33
34
35

```

1. Service (Servis Objeleri)

- **ClusterIP:**

- Varsayılan servis türü.
- Pod'ların **cluster içinde birbirleriyle haberleşmesini** sağlar.
- Dış dünyaya açılmaz.

- **NodePort:**

- Servisi **node'un IP adresi + belirli port** üzerinden dış dünyaya açar.
- Cluster dışından erişim mümkün hale gelir.

- **LoadBalancer:**

- Cloud ortamında **harici yük dengeleyici** sağlar.
- Dış dünyadan gelen trafiği podlara dengeli dağıtır.
- Özellikle production ve cloud ortamında tercih edilir.

2. Liveness & Readiness Probes

- Pod'un **çalışıp çalışmadığını** düzenli olarak kontrol eder.
 - Eğer pod yanıt vermiyorsa Kubernetes otomatik olarak **restart eder**.
 - Örn: Spring Boot "/actuator/health" endpoint'yle canlılık testi yapılabilir.
-

3. Resources (Kaynak Yönetimi)

- **Requests** → Pod'un ayağa kalkması için gereken minimum CPU & RAM.
 - **Limits** → Pod'un maksimum kullanabileceği CPU & RAM.
 - Limit tanımlamak önemlidir: Aksi halde tek bir pod tüm node kaynaklarını tüketebilir.
-

4. Environment Variables

- Pod'lara uygulama ayarlarını (örn: `DB_HOST`, `API_KEY`) eklemek için kullanılır.
 - Tıpkı normal işletim sistemlerinde kullanılan ortam değişkenleri gibi çalışır.
-

5. Secret

- Hassas bilgileri (kullanıcı adı, şifre, API key) **şifreli** şekilde saklar.
 - Pod'lar **valueFrom.secretKeyRef** ile bu bilgilere erişir.
 - Amaç: Pod hacklense bile kritik veriler açıkta olmasın.
-

6. ConfigMap

- Secret gibidir ama hassas olmayan veriler için.
 - Örn: uygulama konfigürasyonu, URL, port.
-

7. Volume

- Pod'lar **kapanınca içindeki veriler silinir** → bu yüzden volume gereklidir.

- Volume, **kalıcı depolama (persistent storage)** sağlar.
 - Pod tekrar başlasa bile veriler korunur.
 - Veri tabanı podları için özellikle kritik.
-

8. Taints, Tolerations & Affinity

- **Taint/Toleration:** Bazı podların sadece belirli node'larda çalışmasını sağlar.
 - Örn: DB sadece "db-node" üzerinde çalışsin, diğer podlar buraya yerleşmesin.
 - **Affinity/Anti-Affinity:** Pod'ların **birlikte ya da ayrı** node'larda çalışmasını kontrol eder.
-

9. Jobs & CronJobs

- **Job:** Tek seferlik iş (örn: DB backup çalıştır).
 - **CronJob:** Belirli aralıklarla çalışan job (örn: her 24 saatte bir yedek al).
 - Kullanım: Yedekleme, batch işlem, periyodik görevler.
-

Özette:

- **Service** = Haberleşme ve erişim.
- **Probe** = Sağlık kontrolü.
- **Resources** = Kaynak sınırlandırma.
- **Secret/ConfigMap** = Güvenli/veri yönetimi.
- **Volume** = Kalıcı depolama.
- **Affinity/Taint** = Pod yerleşim kontrolü.
- **Job/CronJob** = Otomatik işler.

video serisi v4: <https://youtu.be/8Gd4FlxC4ZM?si=sCkf8LTQy0Lcoipi>

The screenshot shows a code editor with a dark theme. The file being edited is named 'DeploymentObject.yaml' under a 'KubernetesYML' project. The code is a YAML configuration for a Deployment object:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgresql-pod
spec:
  replicas: 3
  selector:
    matchLabels:
      database: developer-postgresql
  template:
    metadata:
      labels:
        database: developer-postgresql
      annotations:
        SCM: yazılım
        email: yazılım@bilge.com
        tel: 0 555 999 8877
    spec:
      containers:
        - name: pod-postgresql
          image: postgres
          resources:
            limits:
              cpu: "300m"
              memory: "1024Mi"
            ports:
              - containerPort: 5432
          env:
            - name: POSTGRES_PASSWORD
              value: root
```

Kısa özet: **Deployment**, Docker imajından istenen sayıda **pod** (kopya) oluşturup yöneten, güncelleme/rollback ve ölçekleme işlerini sağlayan Kubernetes objesidir.

Deployment'in işi (kısaca)

- İmajı alır, belirtilen konfig ile pod'ları çalıştırır.
- Replikaları (kopya sayısını) yönetir ve node'lara dağıtır.
- Rolling update / rollback, otomatik yeniden başlatma gibi yaşam döngüsü operasyonlarını sağlar.

YAML'da ana bölümler (ve ne işe yaradığı)

- apiVersion:** `apps/v1` (Deployment için)
- kind:** `Deployment`
- metadata:** `name` , `labels` , `annotations` (metadata bilgi amaçlı; annotation operasyonla etkileşmez)
- spec:** Deployment'a özel ayarlar
 - replicas:** Kaç pod istendiği (ölçekleme)
 - selector.matchLabels:** Hangi pod'ları yöneteceğini belirleyen etiket filtresi
 - bu selector, template içindeki labels ile kesinlikle eşleşmelidir.**

- **template:** Oluşturulacak pod'un şablonu
 - **template.metadata.labels:** Pod'lara atanacak label'lar (servis ve selector için kritik)
 - **template.spec:** Pod içeriği
 - **containers:** `name`, `image`, `ports`, `env`, `resources`, `volumeMounts`,
`liveness/readinessProbe`, `imagePullPolicy`
 - **volumes:** Pod'a bağlanacak PVC / configmap / secret vb.
-

Önemli alanlar — Açıklama & iyi uygulamalar

- **replicas:** Yük arttıkça arttır. Pod'lar bir node'a sıkışmasın; affinity/anti-affinity ile dağıtım kontrolü sağla.
 - **selector ↔ template.labels:** *Bunlar aynı olmalı.* Aksi halde Deployment pod'ları yönetemez.
 - **resources.requests & limits:** Her zaman belirt. (Requests = minimum, Limits = maksimum). Aksi halde bir pod tüm node kaynaklarını tüketebilir.
 - **env:** Hassas verileri doğrudan yazma — `valueFrom.secretKeyRef` ile **Secret** kullan.
 - **Volumes / PVC:** Kalıcı veri için **PersistentVolumeClaim** kullan; DB pod'ları için zorunlu.
 - **probes (liveness/readiness):** Sağlık kontrolleri ekle; Kubernetes gerektiğinde restart veya trafiği keser.
 - **image tag:** `:latest` kullanma — versiyon/semver kullan.
 - **strategy:** Rolling update ayarları (`maxUnavailable`, `maxSurge`) ile kesintisiz deployment sağla.
 - **annotations:** Operasyonel bilgi (sahibi, contact, açıklama) için kullan.
 - **label/field formatı:** Label anahtar ve değerleri küçük harf, kısa ve açık tut; Kubernetes kurallarına uy (örn. `app: my-app`). Büyük harf/camelCase kullanmaktan kaçın.
-

Kısa örnek — Minimal, çalışır durumda bir Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: myrepo/myapp:1.0.0
          ports:
            - containerPort: 8080
      resources:
        requests:
          cpu: "100m"
          memory: "256Mi"
        limits:
          cpu: "500m"
          memory: "1Gi"
      env:
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: password
      livenessProbe:
```

```
httpGet:  
  path: /health  
  port: 8080  
  initialDelaySeconds: 30  
  periodSeconds: 10  
  
volumes:  
  - name: data  
  
  persistentVolumeClaim:  
    claimName: my-pvc
```

Deployment sonrası genelde gerekenler

- **Service** (ClusterIP/NodePort/LoadBalancer) ile dış erişim veya diğer pod'lara erişim sağla.
- **PersistentVolume + PVC** oluştur.
- **ConfigMap / Secret** ile konfig ve hassas verileri yönet.
- Gerekirse **Horizontal Pod Autoscaler (HPA)** ile otomatik ölçeklendir.

Hızlı ipuçları

- `kubectl apply -f deployment.yaml` ile uygula.
- Selector/label uyusuzluğu sık yapılan hata — hep kontrol et.
- Problar + resource limits = stabil sistem.
- Rolling update parametreleri ile production'da kesinti azalt.

video serisi v6: <https://www.youtube.com/watch?v=07P7rjK5GzE&list=PLggmNmVfeLlbICjiTR1AUAF7BhKxNvQzS&index=5>

Terminal: Local

```
muhammetalikaya@Mac-Studio KubernetesYML % kubectl get pods
No resources found in default namespace.

muhammetalikaya@Mac-Studio KubernetesYML % kubectl apply -f DeploymentObject.yml
deployment.apps/postgresql-pod created

muhammetalikaya@Mac-Studio KubernetesYML % kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
postgresql-pod-6d787648c8-c9csw  1/1     Running   0          10s
postgresql-pod-6d787648c8-lkxn5  1/1     Running   0          10s
postgresql-pod-6d787648c8-nvm5b  1/1     Running   0          10s
```

Project: KubernetesYML DeploymentObject.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgresql-pod
spec:
  replicas: 3
  selector:
    matchLabels:
      database: developer-postgresql
  template:
    metadata:
      labels:
        database: developer-postgresql
    annotations:
      SCW: xazilam
```

Terminal: Local (2)

```
muhammetalikaya@Mac-Studio KubernetesYML % kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
postgresql-pod-6d787648c8-c9csw  1/1     Running   0          105s
postgresql-pod-6d787648c8-lkxn5  1/1     Running   0          105s
postgresql-pod-6d787648c8-nvm5b  1/1     Running   0          105s
muhammetalikaya@Mac-Studio KubernetesYML % kubectl get pods -w
NAME          READY   STATUS    RESTARTS   AGE
postgresql-pod-6d787648c8-c9csw  1/1     Running   0          2m3s
postgresql-pod-6d787648c8-lkxn5  1/1     Running   0          2m3s
postgresql-pod-6d787648c8-nvm5b  1/1     Running   0          2m3s
```

The screenshot shows a JetBrains IDE interface with a code editor and a terminal window. The code editor displays a file named `DeploymentObject.yml` containing Kubernetes configuration for a PostgreSQL pod. The terminal window shows the execution of `kubectl get pods` and `kubectl delete pods` commands, followed by another `kubectl get pods` command showing the pods' status.

```

KubernetesYML - DeploymentObject - Bölm11
Project /12 KubernetesYML ~/BILGEADAM_12
> .idea DeploymentObject.yml
  1 metadata:
  2   name: postgresql-pod
  3 spec:
  4     replicas: 3
  5       selector:
  6         matchLabels:
  7           database: developer-postgresql
  8     template:
  9       metadata:
 10         labels:
 11           database: developer-postgresql
 12           annotations:
 13             SCUP: yazilim
 14             email: yazilim@bilge.com
 15             tel: 0 555 999 8877
 16
 17 Document 1/1 spec: template: spec: containers: Item 1/1 resources:
Terminal: Local (2) x + v
muhammetalikaya@Mac-Studio KubernetesYML % kubectl get pods
NAME      READY STATUS RESTARTS AGE
postgresql-pod-6d787648c8-c9csw 1/1 Running 0 105s
postgresql-pod-6d787648c8-lkxn5 1/1 Running 0 105s
postgresql-pod-6d787648c8-nvm5b 1/1 Running 0 105s
muhammetalikaya@Mac-Studio KubernetesYML % kubectl delete pods postgresql-pod-6d787648c8-c9csw
pod "postgresql-pod-6d787648c8-c9csw" deleted
muhammetalikaya@Mac-Studio KubernetesYML % kubectl get pods
NAME      READY STATUS RESTARTS AGE
postgresql-pod-6d787648c8-lkxn5 1/1 Running 0 2m52s
postgresql-pod-6d787648c8-mx92v 1/1 Running 0 9s
postgresql-pod-6d787648c8-nvm5b 1/1 Running 0 2m52s
muhammetalikaya@Mac-Studio KubernetesYML %
muhammetalikaya@Mac-Studio KubernetesYML % kubectl get pods
NAME      READY STATUS RESTARTS AGE
postgresql-pod-6d787648c8-c9csw 1/1 Running 0 2m3s
postgresql-pod-6d787648c8-lkxn5 1/1 Running 0 2m3s
postgresql-pod-6d787648c8-nvm5b 1/1 Running 0 2m3s
muhammetalikaya@Mac-Studio KubernetesYML %
muhammetalikaya@Mac-Studio KubernetesYML % kubectl get pods
NAME      READY STATUS RESTARTS AGE
postgresql-pod-6d787648c8-c9csw 1/1 Terminating 0 2m43s
postgresql-pod-6d787648c8-mx92v 0/1 Pending 0 0s
postgresql-pod-6d787648c8-nvm5b 0/1 Pending 0 0s
postgresql-pod-6d787648c8-c9csw 0/1 ContainerCreating 0 0s
postgresql-pod-6d787648c8-lkxn5 0/1 Terminating 0 2m43s
postgresql-pod-6d787648c8-mx92v 0/1 Terminating 0 2m43s
postgresql-pod-6d787648c8-nvm5b 0/1 Terminating 0 2m43s
postgresql-pod-6d787648c8-c9csw 1/1 Running 0 2s
muhammetalikaya@Mac-Studio KubernetesYML %

```

1. **kubectl** Nedir?

- Kubernetes'e **deklaratif komut** göndermeyi sağlayan araçtır.
- REST API üzerinden çalışır.
- Cloud (AWS, Azure, GCP) kullanıyorsan ek konfigürasyon gerekebilir, ama Docker Desktop gibi lokal ortamda direkt kullanılabilir.

2. Temel Komutlar

◆ **kubectl get**

- Objeler hakkında bilgi almak için kullanılır.
- Örnekler:

```

kubectl get pods      # Podları listeler
kubectl get pods -o wide # Podları detaylı listeler (IP, node bilgisi vs.)

```

```
kubectl get nodes      # Node'ları listeler  
kubectl get nodes -o wide # Node detayları
```

◆ **kubectl apply -f**

- YAML dosyasını kullanarak obje oluşturur/günceller.
- Örn:

```
kubectl apply -f deployment.yaml
```

◆ **kubectl delete**

- Belirtilen objeyi siler.
- Örn:

```
kubectl delete pod <pod-adı>
```

◆ **kubectl describe**

- Belirtilen obje hakkında **detaylı bilgi** verir.
- Örn:

```
kubectl describe pod <pod-adı>
```

◆ **kubectl get ... --watch**

- Objeleri **canlı olarak izler**.
- Örn:

```
kubectl get pods --watch
```

3. Deployment ve Pod Mantığı

- Deployment → "Her zaman şu kadar pod çalışsın" şeklinde **deklaratif tanım** yapar.
- Örneğin: "3 replica PostgreSQL pod çalışsın" dediğinde, biri silinse bile otomatik olarak yenişi ayağa kalkar.
- Silinen pod → terminate edilir, aynı anda yenişi create edilir.

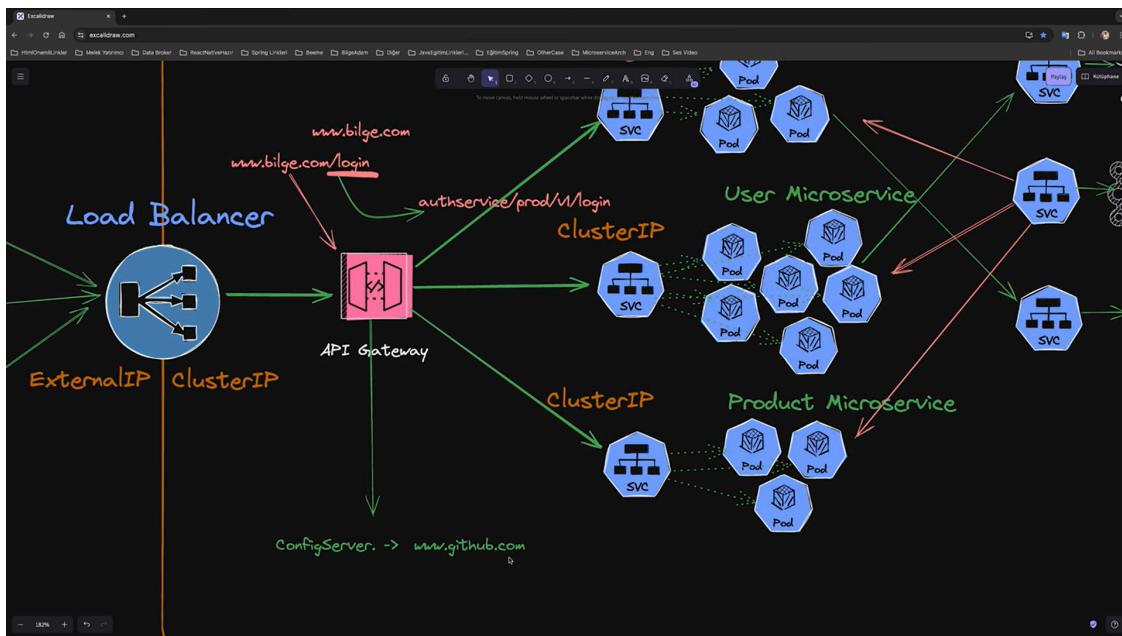
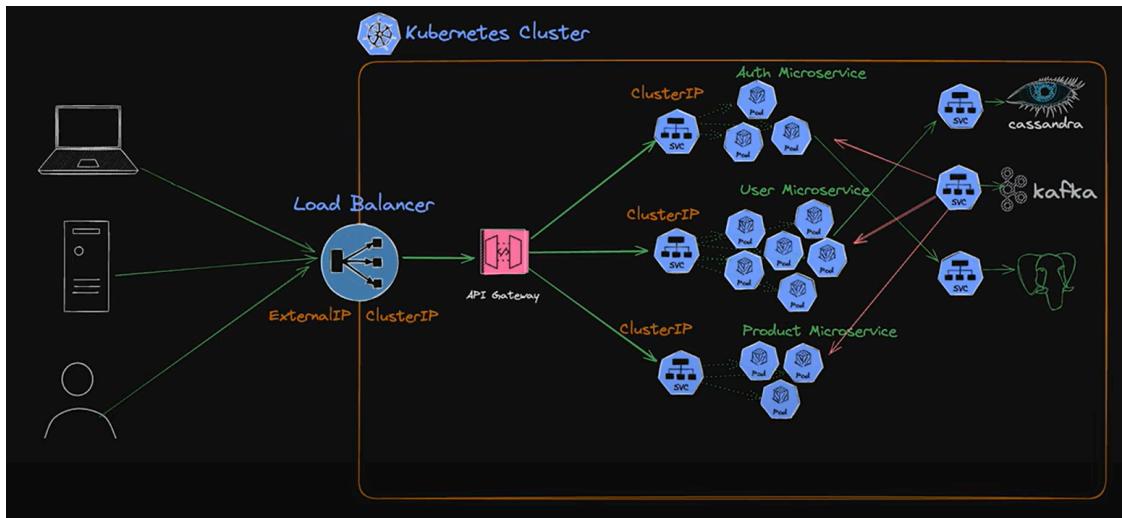
4. Describe Çıktısında Görülebilenler

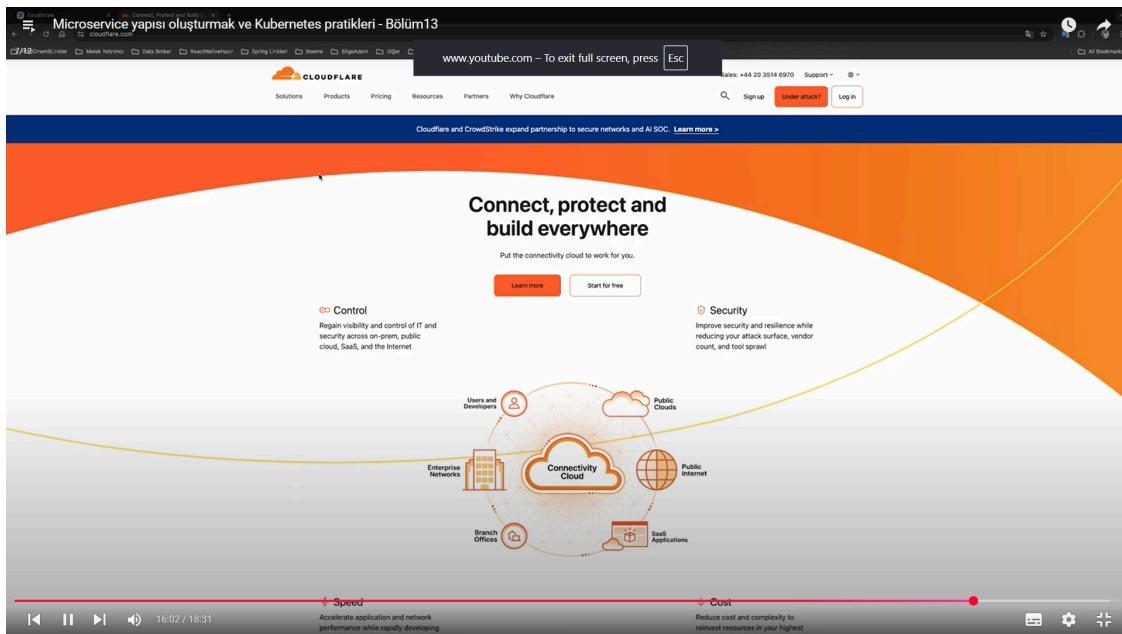
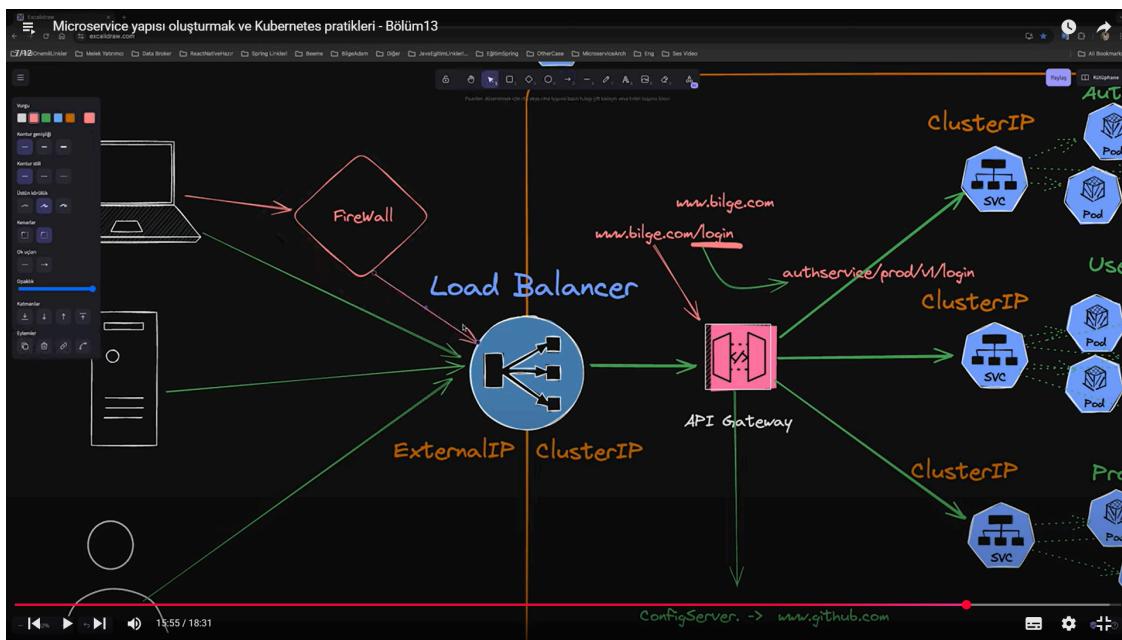
- Pod adı
- Namespace
- Label & Annotation (kim oluşturmuş, açıklamalar vs.)
- IP adresi
- Container imajı ve versiyonu
- Event geçmişi

✓ Özeti:

- **kubectl** ile Kubernetes objelerini **oluşturur, siler, listeler, izler ve incelersin.**
- **Get** = Listele
- **Apply** = Oluştur/Güncelle
- **Delete** = Sil
- **Describe** = Detaylı inceleme
- **Watch** = Canlı takip
- Deployment sayesinde podlar hep ayakta tutulur.

video serisi v7: <https://www.youtube.com/watch?v=F-JvOvD19zk&list=PLggmNmVfeLlbICjiTR1AUAF7BhKxNvQzS&index=7>





1. Temel Yapılar

- **Cluster:** Kubernetes'in çalıştığı ortam (Docker Desktop, Google Cloud, AWS, Azure olabilir).
- **Node:** Fiziksel/virtual makine gibi düşünülebilir.
- **Pod:** Node içinde çalışan en küçük birim. İçinde container(lar) barındırır.

- **Container**: Docker image'inden oluşturulan çalıştırılabilir ortam.

👉 Genelde **1 pod = 1 container** tercih edilir.

2. Deployment

- Kodlarımızı (ör. Spring Boot, mikroservis) **image** haline getiririz.
 - Bu image → Docker Hub'a yüklenir → Kubernetes podları bu imajı indirir.
 - Podları yönetmek için **Deployment objesi** kullanılır.
-

3. Service

- Pod IP'leri **değişkendir** (restart, güncelleme vs. yüzünden).
- **Service objesi**:
 - Pod'ların önüne geçer.
 - **Load balancer görevi görür**.
 - Tek bir sabit endpoint üzerinden podlara ulaşılmasını sağlar.

👉 Hem mikroservisler hem de veritabanları (Postgres, Cassandra, Kafka vs.) için şarttır.

4. API Gateway

- Yüzlerce mikroservisin tek tek IP'siyle uğraşmamak için kullanılır.
 - Gelen istekleri **doğru servise yönlendirir**.
 - Örn: `/login` → Auth service, `/products` → Product service.
 - Config bilgilerini **Config Server** (GitHub repo vs.) üzerinden alabilir.
-

5. Dış Dünyaya Açılmaya

- **ClusterIP** → Cluster içi iletişim.
- **NodePort** → Node'un portunu dışa açar.

- **LoadBalancer** → Harici IP sağlar, dış dünyadan erişim mümkün olur.

👉 LoadBalancer dış dünyadan gelen trafiği cluster içindeki servislere yönlendirir.

6. Güvenlik Katmanı

- Load balancer önüne **firewall** eklenebilir.
- Cloudflare gibi servisler:
 - Gerçek IP'yi gizler.
 - DDoS, bot saldırısını engeller.
 - Domain → Cloudflare → Firewall → LoadBalancer → Gateway → Service → Pod zinciriyle çalışır.

✓ Genel Akış:

Kullanıcı → **LoadBalancer** → **API Gateway** → **Service** → **Pod (Container)**

Dockerfile, Docker Compose, Kubernetes farkı ne bir projede hepsi olmak zorunda mı

1 Dockerfile

- **Ne işe yarar?** Uygulamanızı bir **container image**'e dönüştürür.
- **İçerik:** Uygulama kodu, bağımlılıklar, ortam değişkenleri, çalışma dizini, komutlar...
- Örnek: Python uygulamanız varsa, Dockerfile ile hangi Python versiyonu, hangi kütüphaneler kullanılacak, uygulama nasıl çalışacak belirlenir.
- **Kullanım:** `docker build` ile image üretirsiniz.

2 Docker Compose

- **Ne işe yarar?** Birden fazla container'ı **birlikte çalıştırma**k için kullanılır.
 - **İçerik:** Hangi container hangi image'den çalışacak, port mapping, volume mount, environment variables, network tanımları...
 - Örnek: Uygulamanızın backend, frontend ve veritabanı container'ları varsa, Compose ile hepsini tek komutla başlatabilirsiniz ([docker-compose up](#)).
 - **Avantaj:** Lokal geliştirme ve test ortamını hızlı kurmak için ideal.
-

3 Kubernetes (K8s)

- **Ne işe yarar?** Container'ları **büyük ölçüde yönetir**.
 - **İçerik:** Deployment, Service, Ingress, ConfigMap gibi objelerle container'larınızın otomatik olarak ölçeklenmesini, load balancing, restart policy vs. sağlar.
 - **Kapsam:** Genellikle üretim ortamı için kullanılır. Binlerce container'ı yönetebilir.
 - **Avantaj:** High availability, self-healing, rolling update gibi enterprise seviyesinde özellikler sunar.
-

Özet fark

Özellik	Dockerfile	Docker Compose	Kubernetes
Amacı	Image oluşturmak	Multi-container çalıştırma	Büyük ölçüde container yönetimi
Kullanım	Tek image	Local dev / test	Prod / ölçeklenebilir ortam
Öğrenme seviyesi	Temel	Orta	İleri

Hepsi bir projede zorunlu mu?

- **Hayır.**
- Küçük bir projede sadece Dockerfile yeterli olabilir.
- Birden fazla container varsa, Compose çok yardımcı olur.
- Büyük ölçekli ve prod ortamı için Kubernetes mantıklı.

 Örnek senaryolar:

- **Basit Python script**: Dockerfile yeterli.
- **Web app + DB (local dev)**: Dockerfile + Docker Compose.
- **Üretim ortamı, ölçülebilir mikroservisler**: Dockerfile + Kubernetes (Compose lokal dev için opsiyonel).

