kubernetes-udemy-notes

*Kubernetes, dağıtık mimarideki yapıları production ortamlarında kesintisiz çalıştırabilmemiz için stabil ortam sunar.

=> Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem.
=>Kubernetes hem beyan temelli(declarative) yapılandırmayı hem de otomasyonu kolaylaştıran, container iş yüklerini ve hizmetleri yönetmek için oluşturulmuş, taşınabilir ve genişletilebilir açık kaynaklı bir platformdur.

==>>KUBERNETES COMPONENTLERİ

A) CONTROL PLANE(Master Node)
1-) kube-apiserver:
-diğer tüm component ve node bileşenlerinin direkt iletişim kurabildiği tek component.
-tüm iletişim apiserver üzerinden gerçekleşir
-control plane' nin en önemli bileşeni ve giriş noktası, (resepsiyon)
-kullanıcılar kubectl komut satırı ve Rest-api çağrıları aracılığıyla kube-apiserver ile iletişime geçebilirler.

2-) etcd
-tüm cluster verisi, metada bilgileri ve oluşturulan tüm objelerin bilgilerinin "key-value" olarak tutulduğu veri deposudur.
-kubernetesin çalışması için gerekli tüm veriler burada tutulur.
-etcd ile sadece kube-apiserver doğrudan haberleşebilir. Diğer bileşenlerde sadece apiserver aracılığıyla etce ile iletişim kurabilir.

3-) kube-scheduler
- podların hangi node da çalışacağını belirler.
-yeni oluşturulan veya herhangi bir node ataması yapılmamış pod' ları takip eder, özel gereksinimlerini inceler, elindeki parametre ve algoritmalara göre en uygun node 'a karar verir.

4-) kube-controller-manager
-tek bir binary dosyası olarak derlenmiştir.
-içerisinde birden fazla controller vardır

-tek bir process olarak çalışır.
-istenilen durum ile şimdiki durumu karşılaştırmak için kullanılır.
-bu controllerlar kube-apiserver aracılığıyla etcd' de saklanan cluster durumunu gözler istenilen-mevcut durum farkı varsa bu farkı oluşturan kaynakları siler/oluşturur/günceller.
- Node controller
- Job controller
- Service account & Token controller
- Endpoints controller
* Cloud controller

B)WORKER NODE
*worker node'lar podların üzerinde çalıştığı yerlerdir.
*iş yükleri worker nodelarda bulunur.
*worker node' lar üzerlerinde container runtime barındıran ve cluster'a dahil edilen sistemlerdir.
*her worker node' da 3 temel component bulunur.
*Clusterdaki her node çalışan bir agent'tır ve pod içerisinde tanımlanan containerların çalıştırılmasını sağlar.

1-) Container Runtime
-containerları çalıştırmaktan sorumlu yazılım, docker, containerd, cri-o...

2-) kubelet
-apiserver aracılığıyla etcd'yi kontrol eder ve scheduler tarafından kubeletin bulunduğu node üzerinde bir pod çalıştırılması talimatı verildiyse kubelet bu pod'u bulunduğu node'da yaratır. Containerd'ye haber gönderip belirlenen özelliklerde container'ın o sistemde çalışmasını sağlar.
-primary node agent
-ensures container are running and healthy

3-) kube-proxy
-oluşturulan podların tcp-udp-lctp trafik akışlarını yönetir, ağ kurallarını belirler-yönetir.
- lojistik elemanı,
-network proxy görevi görür.

==>> Kubernetes Versiyonları:
-x.y.z - major.minor.patch
-senede 3 minor versiyon
-her minor versiyon 12 ay desteklenir.

kaynaklar ** cncf.io, kubernetes.io, github.com/kubernetes/kubernetes

==>>KUBECTL
==>>kubectl config dosyası:
-kubectl default olarak $HOME/.kube/ altındaki config isimli dosyaya bakar
-KUBECONFIG environment variable değerini değiştirerek bunu güncelleyebiliriz
-config dosyasında cluster bağlantı bilgilerini ve kullanıcıları belirleriz.
-hangi cluster'a hangi user ile bağlanacağımızı context bölümünde belirleriz.

$ kubectl config get-contexts
> mevcut contextleri listeler, * ile current context'i gösterir.

$ kubectl config current-context
> current context'i görmek için

$ kubectl config use-context minikube
> current context değiştirmek için

$ kubectl cluster-info
> cluster hakkında bilgi verir

$ kubectl get pods -n kube-system
> -n opsiyonuyla namespace belirtiriz

$ kubectl get pods -A
> tüm namespacelerdeki podları getir

$ kubectl get pods -A -o wide
> output formatını değiştirmek için
wide, yaml, json

$ kubectl get pods -A -o json | jq -r ".items[].spec.containers[].name"

$ kubectl explain pod-node.....
>pod, node gibi objeler hakkında bilgi almak için
>hangi api zerinde durduğunu öğrenmek için

==>> POD
-en küçük temel birim/obje
-bir veya birden fazla container barınabilir.(çoğunlukla bir container)
-her pod'un ayrı bir id' si olur(uniq identifier)
-her pod uniq bir ip ye sahiptir.

$ kubectl run firstpod --image=nginx --restart=Never
$ kubectl get pods -o wide
$ kubectl describe pods firstpod
>obje hakkında ayrıntılı bilgi edinmek için

$ kubectl logs firstpod
>loglara bakmak için

$ kubectl logs -f firstpod
>log kayıtlarını anlık/canlı takip edebilmek için

$ kubectl exec firstpod -- hostname
>pod üzerinde komut çalıştırmak için

$ kubectl exec -it firstpod -- /bin/sh
>pod 'a bağlanmak için

$ kubectl delete pods firstpod
>podu silmek için

$ kubectl explain pods/deployments
> kubernetes objelerinin api versiyonlarını öğrenmek için

apiVersion: v1
kind: Pod
metadata: > objeyle ilgili unique bilgilerin tanımlandığı yer: **name, labels, annotations** gibi objeye özel bilgiler tanımlanır, string değildir, dictionary şeklinde tanımlanması gerekir.
spec: > her obje tipine göre değişir, objenin özellikleri tanımlanır, bu tanımlar dokümantasyondan bulunur.

**consolde dan komut ile pod oluşturunca bu imperatif bir yöntemdir, çalışan bir podu bu yöntemle güncellemek daha uzun sürer çünkü komut satırından silip yeni komut girmek gerekir. Bu nedenlerle deklaritif yöntem kullanmak yani yaml dosyasıyla yapmak çok daha

kolay ve efektifdir.

kubectl edit
-herhangi bir k8s objesi üzerinde değişiklik yapmak istediğin zaman kullanabileceğin bir komuttur. Default text editorunde bir yaml dosyası açar. çok sık kullanılmaz, versiyon kontrolu de sağladığı için yaml dosyası üzerinde değişiklik yapmak best practice dir. edit troubleshoot yaparken kullanılır. etkileri kalıcı olmaz.
$ kubectl edit pods firstpod

POD OLUŞTURMA YÖNTEMLERİ
1-kubectl ile komut satırından->imperatif yöntem
2-pod.yaml dosyası ile apply ederek-> deklaratif

Eğer oluşturduğumuz bir pod uzun süre pending aşamasında takılı kalıyorsa bu durum kube-scheduler'ın podun oluşturulması için uygun bir node bulamadığı anlamına gelir.

***tüm nodelarda kubelet bulunur, bu kubelet' ler bulundukları node' a bir pod/obje atanıp atanmadığını görmek için sürekli etcd' yi gözler, yeni atanmış bir node görürse pod tanımında containerlar içinde çalışması istenen imajlara bakar ve bunları indirir. imaj indirilemezse öncelikle err image pull hatası devamında image backoff hatası verir.

***Eğer pod'un status kısmında image pull backoff hatası görürseniz bu node'un imajı repository'den çekemediği ve tekrar tekrar denemeye devam ettiği anlamına gelir. İmaj adını yanlış yazılması, düzgün login olunmaması gibi durumlarda bu hatayı alabiliriz.

RESTART POLICY
-Always
-On-Failure
-Never

*Pending->Creating->ImagePullBackOff->Running-> Succeeded/Completed/Failed->CrashLoopBackOff

CrashLoopBackOff-> oluşturduğun pod tekrar tekrar kapanıp yeniden oluşturuluyor bir problem olabilir anlamına gelir ve bir şeylerin ters gittiğinin göstergesidir. Sürekli restrart edilen bir container var.*

==>> yaml dosyasında command opsiyonunun kullanımı:

```
apiVersion: v1
kind: Pod
metadata:
  name: succeededpod
spec:
  restartPolicy: Never
  containers:
  - name: succeedcontainer
    image: ubuntu:latest
    command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 20']
```

==>> Multi Container Pod- Çoklu Container Pod (30.Bölüm)

*Wordpress-> PHP tabanlı bir frontend katmanına ve verilen tutulduğu bir mysql veritabanına sahiptir.
**Bu iki uygulama teknik olarak tek bir containerda çalıştırılabilir ancak bu tercih edilmez çünkü uygulamaların izole çalışmasını isteriz.
***Ayrıca uygulamanın scale edilmesi gereken durumlarda, ideal şekilde yatay büyüme yapamıyoruz.
****Yalnızca helper uygulamaların aynı containerda/Podda çalışması uygundur(bağımlı uygulamalar).
=>Kubernetes, ana uygulamaya bağımlı ve onunla network seviyesinde izolasyon olmadan ve gerekirse aynı depolama altyapısını kullanabilecek uygulamaları pod içerisinde ikinci bir container olarak çalıştırma imkanı sağlar.(Sidecar container)
-Aynı pod içinde bulunan containerlar birbirleriyle local host üzerinden anlaşabilirler.(network izolasyonu yoktur.)
-Tek bir volume yaratılarak aynı pod içerisindeki containerlara mount edilebilir ve aynı dosyalar üzerinde çalışılabilir.

```
apiVersion: v1
kind: Pod
metadata:
  name: multicontainer
spec:
  containers:
  - name: webcontainer
    image: nginx
    ports:
      - containerPort: 80
    volumeMounts:
```

```
    - name: sharedvolume
      mountPath: /usr/share/nginx/html
  - name: sidecarcontainer
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do wget -O /var/log/index.html https://
raw.githubusercontent.com/ozgurozturknet/hello-world/master/
index.html; sleep 15; done"]
    volumeMounts:
    - name: sharedvolume
      mountPath: /var/log
  volumes:
  - name: sharedvolume
    emptyDir: {}
```

$ kubectl apply -f podmulticontainer.yaml
=> pod içindeki containerlardan herhangi birine bağlanabilmek için;
$ kubectl exec -it multicontainer -c webcontainer -- /bin/sh
=>container'ın adını görmek için;
hostname

=>Aynı pod içinde çalışan containerlarda network izolasyonu olmadığı için aynı ip adreslerini alırlar, yani aynı makinedeymiş gibi çalışırlar
-aynı volume mount edildiğinden her iki containerdaki volumeler aşağıdaki pathlerden incelenebilir.
cd /usr/share/nginx/html
cd /var/log

=> logları incelemek için;
$ kubectl logs -f multicontainer -c sidecarcontainer

==>> Init Container

*Uygulama container başlatılmadan önce ilk olarak init container çalışır.
*Init container yapması gereken işlemleri tamamlar ve kapanır
*Uygulama containerı init container kapatıldıktan sonra çalışmaya başlar.
*Init container işlemlerini tamamlamadan uygulama containerı başlatılmaz.

==>>> Labels and Selectors(bölüm 33)

label ->kubernetes de her türlü objeye atanabilen key-value değerleridir.(aidiyet ataması-tanımlama-gruplama ve objeler arasında ilişki kurulmasında kullanılırlar)

example.com/tier:front-end
prefix key:value

**prefix(önek) kısmı opsiyoneldir,
**kubernetes.io/ ve k8s.io/ prefixleri kubernetes için bileşenleri için ayrılmıştır.
**key-value -> en fazla 63 karakter olabilir ve alfanumerik karakterlerle başlayıp bitmelidir(a-z A-Z 0-9)

** podları label atama metadata kısmında yapılır.
** bir poda aynı key iki kere atanamaz, örneğin app adına sadece bir key verebiliriz.

$ kubectl get pods -l “app” --show-labels
$ kubectl get pods -l “app=firstapp”

**eşitlik temelli selector kullanımı -equality based
$ kubectl get pods -l “app=firstapp,tier=frontend”

**atama temelli label kullanımı ->setbase
$ kubectl get pods -l ‘app in (firstapp)’ --show-labels
$ kubectl get pods -l ‘app in (firstapp, secondapp) --> app anahtarına firstapp VEYA secondapp değeri atanmış podları gösterir.
$ kubectl get pods -l ‘app not in (firstapp, secondapp)
$ kubectl get pods -l “app in (firstapp), tier notin (frontend)” --show-labels

=> terminalden label verme
$ kubectl label pods pod9 app=thirdlabel
=> terminalden label silme
$ kubectl label pods pod9 app-
=> terminalden pod label güncelleme
$ kubectl label --overwrite pods pod9 team=team3
=> namespacedeki tüm podlara label verme
$ kubectl label pods --all foo=bar

=> Annotation

*Podlara aynı labeller gibi key=value şeklinde metadata ekleyebilmenin bir yoludur.
*labeller hassas data olarak görülür, yani bir label ekleme ya da çıkarma kubernetes içinde bir şeyleri tetikleyebilir.
**bu nedenle her bilginin label olarak atanması doğru değildir, bu gibi durumlarda ve özellikle selector ile filtreleme yapmayacaksak annotation kullanılır.

**mesala podun oluşturma tarihini veya oluşturanın kimliğini metadata olarak eklemek istersek bunu label yerine annotation olarak ekleriz.
--özetle selector ile filtreleme yapmayacaksak veya objeler arası iletişim için kullanmayacaksak metadataları annotation ila pod'a iliştiririz.

**kubernetes ile bağlantısı olan yazılımların ihtiyaç duyabileceği bilgilerde annotation olarak kaydedilir.

--annotation value kısmına her türlü karakter yazılabilir.

```
apiVersion: v1
kind: Pod
metadata:
  name: annotationpod
  annotations:
    owner: "Ozgur OZTURK"
    notification-email: "admin@k8sfundamentals.com"
    releasedate: "01.01.2021"
    nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
spec:
  containers:
  - name: annotationcontainer
    image: nginx
    ports:
    - containerPort: 80
```

==> annotationları görmek için için;
$ kubectl describe pod <podname>

==> imperative olarak annotation ekleme;
$ kubectl annotate pods <podname> foo=bar
==> imperative olarak annotation silme;

$ kubectl annotate pods <podname> foo-

==>>NAMESPACE - (bölüm 36)
*Namespaceler cluster kaynaklarını birden çok kullanıcı arasında bölmenin yoludur.

-Namespaceler adlar için bir kapsam sağlar
-Kaynak adlarının bir namespace içinde benzersiz olması gerekir
-Namespace birbirinin içine yerleştirilemez, (nested olamaz)
-her kubernetes resource yalnızca bir namespae içinde olabilir.

$ kubectl get namespaces

**namespace de bir obje olduğundan yaml dosyaları ile oluşturulabilir.
***diğer nesnelerinde hangi namespace altında oluşacağını belirtmek istiyorsak metadata altında namespace başlığı kullanırız.

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
---
apiVersion: v1
kind: Pod
metadata:
  namespace: development
  name: namespacepod
spec:
  containers:
  - name: namespacecontainer
    image: nginx:latest
    ports:
    - containerPort: 80
```

**varsayılan olarak kubectl komutları default namespace de uygulanır.

$ kubectl get pods --namespace kube-system
$ kubectl get pods --all-namespaces veya -A

=> terminalden namespace oluşturmak
$ kubectl create namespace app1
$ kubectl get pods -n development

=>farklı namespace deki poda bağlanmak için;
$ kubectl exec -it <podname> -n <namespace> -- /bin/sh

==> varsayılan namespace'i değiştirmek;
$ kubectl config set-context --current --namespace=development

=>namespace silmek;
****namespace silindiğinde o namespace'deki tüm objeler silinir**, o yüzden dikkatli olmak gerekir.
$ kubectl delete namespaces development

==>DEPLOYMENT

Deployment, bir veya birden fazla podu bizim belirlediğimiz desired state'e göre oluşturan ve sonrasında bu desired state mevcut durum ile sürekli karşılaştırıp gerekli düzeltmeleri yapan bir obje tipidir.

** Biz deployment'da istenen durumu tanımlarız ve deployment controller mevcut durumu istenilen durumla karşılaştırıp gereğini yapar.

İmperative olarak deployment oluşturma;
$ kubectl create deployment firstdeployment --image=nginx:latest --replicas=2

=> mevcut deploymenttaki imajı güncellemek;
$ kubectl set image deployment/firstdeployment nginx=httpd

==> pod sayısını değiştirmek
$ kubectl scale deployment firstdeployment --replicas=5

=> deployment silmek
$ kubectl delete deployment firstdeployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: firstdeployment
  labels:
    team: development
spec:
```

```
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
```

**deploymentlarda en az bir tane selector(spec.selector kısmında) olmalı ve template.matedata.labels kısmı ile uyuşmalı

=> ReplicaSet
-genellikle belirli sayıda özdeş Pod'un kullanılabilirliğini garanti etmek için kullanılırlar.
-bir deployment oluşturduğumuzda bu deployment objesi kendi yönettiği bir replicaSet objesi oluşturur, bu replicaSet objesi de podları oluşturur ve yönetir. mesala kullanılan imaj güncellenirse yeni imaj ile yeni replicaSet oluşur ve yeni replicaSet yeni podlar oluştururken eski replicaSet de önceki podları siler, bu oluşturma ve silme işlemlerinin sırasını biz belirleyebiliriz.

$ kubectl set image deployment/firstdeployment nginx=httpd
=> Son değişikliği geri alma;
$ kubectl rollout undo deployment firstdeployment

**replicaSet objesi, bize deployment objesinin sunduğu rollout, undo gibi olanakları sunmaz, replicaSet yaml dosyasında değişiklik yapsak, mesela imaj değiştirsek ve apply desek yeni imajdan pod oluşturmaz, bu nedenle biz deployment kullanırız.

=>ROLLOUT - ROLLBACK
*Deployment objesinin altında spec.strategy.type= **Recreate** şeklinde rollout stratejisi set edersek bu "ben bu deployment da bir değişiklik yaparsam öncelikle tüm mevcut podları sil ve bu işlem tamamlandıktan sonra yeni podları oluştur" anlamına gelir.
-Bu stratejiyi genellikle hardcore migration yaptığımız durumlarda kullanırız. Ör.major değişikliklerde kullanılır.
-recreate esnasında küçük kesintiler olabilir. Zaten iki versioyonun aynı anda çalışmasını istemiyorsak bunu kullanırız.

*Mevcut **deployment'daki imajı değiştirmek** için aşağıdaki komutu kullanırız.
$ kubectl set image deployment rcdeployment nginx=httpd

*Deployment objesinin altında spec.strategy.type= **RollingUpdate** şeklinde bir rollout stratejisi kullanırsak bu "beni bir değişiklik yaptığım zaman verdiğim maxUnavailable ve maxSurge parametrelerine göre aşamalı olarak deployment'ı güncelle" demiş oluyoruz.
-bu strateji zero downtime ile yeni deployment a geçiş yapmamıza olanak verir.
maxUnavailable: geçiş sırasında aynı anda en fazla kaç pod sileceği belirtilir, burada yüzde de verilebilir.
maxSurge: geçiş sırasında toplam pod sayısının en fazla kaç olabileceğini belirlemek için kullanılır.
-Default olan strateji RollingUpdate seçilidir. %25 olarak ayarlar set edilmiştir.

$ **kubectl apply -f deployrolling.yaml --record**

-deployment'ı güncellemek istiyorum.(bu sefer set komutu yerine edit kullanıyorum)
$ kubectl edit deployment rolldeployment --record

***-- record** opsiyonu sıkıntı olursa başladığımız yere geri dönebilmek için kayıt oluşturmaya yarar. kubectl rollout history deployment <deploymentname> komutuyla record kayıtlarını görebilir, kubectl rollout history deployment <deploymentname> --revision=2 komutuyla istenen güncellemede ne değişikliği yapıldığı gözlemlenebilir.

*bir önceki versiyona geri dönmek istersem:
$ **kubectl rollout undo deployment <deploymentname>** komutunu kullanabilirken, komutun sonuna **--to-revision=1** opsiyonunu eklersem istediğim versiyona dönebilirim.

*kubectl rollout **status** deployment <deploymentname>
*status opsiyonu canlı olarak deploymentın oluşturulma durumun gösterir.
kubectl rollout **pause deployment <deploymentname>
*pause opsiyonu rollout komutu ile güncelleme yapılırken, sistemde bir problem olduğu düşünülürse bu komut girilerek deployment rollout durdurulur, (pause edilir), problemin ne olduğunun tespit edilebilmesi için kullanılabilir, devam ettirmek için ise kubectl rollout resume deployment "deployname" komutu kullanılır.

**bu konuya probeları işlerken tekrar döneceğiz.

==>KUBERNETES AĞ ALTYAPISI<==

*K8s Ağ Kuralları:
-pod'lara ip adres aralığı belirlenir. --pod-network-cidr
-her pod bu cidr bloğundan atanacak eşsiz bir ip adresine sahip olur.
-aynı cluster içerisindeki tüm podlar varsayılan olarak birbirleriyle herhangi bir kısıtlama olmadan ve NAT(network adres translation) olmadan haberleşebilirler.

CNI(container network interface)
CNI, containerların ağ bağlantısıyla ve containerlar silindiğinde ayrılan kaynakların kaldırılmasıyla ilgilenir. Network plug-in kullanılmasına imkan veriyor, k8s de bunu destekledi,
SonuçOlarak:**Kubernetes kurulumunda ortama uygun bir CNI network plug-in seçmek zorundayız.**

Son dönemde bu pluginlerden iki tanesi çok öne çıkmış.
1-**flannel:**
2-**calico:** network policy gibi ek özellikler sunar

-->>bu iki pluginin den birini seçerek network yönetimini bu plugine devretmiş oluyoruz.
-podlara ip adreslerinin atanması,
-ip table kurallarının düzenlenmesi
-node'lara arası NAT kullanmadan çalışabilme(overlay)
gibi tüm görevler calico ya da flannel gibi plug-in'ler tarafından yapılıyor.

==>> SERVICE <<==

-Kubernetes ortamlarında podlar sürekli oluşturulup silindiğinden statik bir ip adresi kullanılması mümkün değildir, service objesi bu sıkıntıyı aşmak için kullanılır.

*4 tip service objesi oluşturulabilir.
1-ClusterIP: Only reachable from within the cluster.(default), basit de olsa kubernetes altında service discovery ve load balancing hizmeti veren obje tipidir
2-NodePort: From outside the cluster (30000-32767) tüm worker

node'larda bu port sizin servisinize prox'lenir, yani worker node'un ip adresinin belirlediğimiz portuna gelen tüm paketler, ilişkilendirdiğimiz podların publish ettiğimiz portlarına yönlendirilir.
3-LoadBalancer: cloud service sağlayıcılarının managed kubernetes servicelerinde kullanılır. Örn.AWS dış dünyadan erişilebilecek bir loadbalancer oluşturur ve service adresiyle ilişkilendirilir.
4-ExternalName: is a special case of service that does not have selectors and it uses DNS names instead. externalName maps the Service to the contents of the externanName field by returning a CNAME record with its value.(yani bir servisi DNS name ile eşleştirmede kullanılır.)

=>Service objelerinin listelenmesi
$ kubectl get service

=>Bir deployment'in **expose** edilmesi "imperative olarak service objesinin oluşturulması"
$ kubectl expose deployment "deployment_ismi" --type="service_tipi" --name="servis_ismi"
Ör: kubectl expose deployment backend --type=ClusterIP --name=backend

=>Service objelerinin silinmesi
$ kubectl delete service "servis_ismi"
Ör: kubectl delete service backend

=>service.yaml örneği

```
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  type: ClusterIP
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
```

*service hangi pod'lara hizmet edeceğini selector ile seçiyor
port:bu servisin dinleyeceği port

targetPort: podların expose ettiği port

**Pod'a bağlanmak için
$ kubectl exec -it <podname> -- bash

Nodeport u denemek için tünel açan komut(starting tunnel for service frontend)
**minikube service --url frontend

***biz bir service oluşturduğumuzda, bu service **endpoint** adında bir obje oluşturur.
***bu endpoint objesi bizim belirlediğimiz selector tarafından seçilen podların IP adreslerini üzerinde barındırır.
***Service trafiği nereye yönlendireceğini bu listeye göre düzenler
$ kubectl get endpoints

->deployment'ı **scale** etme komutu
$ kubectl scale deployment <deploy.name> --replica=5

=> LIVENESS PROBES

*Bazen container ayakta ama uygulama çalışmıyor olabilir. Bu durumları kontrol etmek ve container'ın ne zaman yeniden başlatılacağına karar verebilmek için livenessprobe kullanılır.
Bunu çeşitli yollar ile yapabilir:
1-Container içerisinde komut çalıştırarak(exec)-shell de komut ya da uygulama çalıştırırız, 0 yani olumlu kod dönerse sağlıklı olduğunu anlarız, başka kod dönerse pod'u yeniden başlatırız.

2-http endpoint'e request göndererek(httpGet)-endpoint uygulamaya özel olarak kurgulanmalı

3-bir port'a tcp connection açarak (tcpSocket)- tcp port ile sorgulama yapılır. açık olan porta istek gönderir olumsuz bir yanıt alırsa pod restart edilir.

==> READINESS PROBE

*Pod'un loadbalancer servisinden hizmet almaya hazır olduğunu, yani tam anlamıyla ayakta ve çalışır vaziyette olduğunu anlamak için kullanılır.
**Kubelet, bir container'ın ne zaman trafiği kabul etmeye hazır

olduğunu bilmek için readiness probeları kullanır.
***Bir por, tüm container'lar hazır olduğunda hazır kabul edilir.
****Readiness probe sayesinde bir pod hazır olana kadar service arkasına eklenmez.

-liveness probe da olduğu gibi exec, httpGet, tcpSocket methodları readiness probeda da uygulanabilir.
-Check sonuçları olumlu dönüyorsa pod servise eklenir.
Özetle:readiness probe, pod'un servis sunmaya hazır halde olduğunu belirler.

-> **terminationGracePeriodSeconds**:30
Şu anda yaptığın işlem varsa 30 saniye içinde tamamla ve sonra kapan demek için kullanılır. 30 saniyeden sonra zorla kapanır

**pod'a exec komutuyla bağlanıp dosya silmek
$ kubectl exec <podname> -- rm -rf <filename>

=>Resource Limits
-CPU
*kesirli değerlere izin vardır.
cpu:"1"="1000"="1000m" aynı anlama gelir(m=milliCPU)
cpu'nun %10' unu kullansın dersek
cpu:"0.1"="100" veya "100m" olarak ifade ederiz.
-MEMORY
*byte cinsinden tanımlanır.
ki, mi, gi olarak da kullanılabilr.

-requests : pod'un oluşturulabilmesi için minimum ne kadarlık boş kaynağın olması gerektiğini belirtir.

-limits : container'ın kullanabileceği en fazla sistem kaynağını set etmek için kullanılır.

**pod belirlenenden fazla cpu kaynağı kullanamaz.
***memory'de durum farklı, pod set edilen memory'den fazla kullanmak isterse bunu engelleyen prosedür olmadığından OOMKilled(out of memory killed) durumuna geçer ve restart edilir.

==>>ENVIRONMENT VARIABLES <<==
spec:
  containers:

env:
-name: USER
value: “talha”

**pod’a bağlanıp içindeki env var’ları görmek için:
$ kubectl exec podname -- printenv

***ben kubectl aracılığıyla herhangi bir poda, deploymenta ya da servise kendi bilgisayarımdan tünel açarak o objenin portuna trafiğimi yönlendirebilirim. Testler esnasında objelerimize hızlı bir şekilde bağlanabilmek için kullandığımız bu özelliği **port forwarding** deniyor.
$ kubectl port-forward object/objectname hostport:containerport
$ kubectl port-forward pod/envpod 8080:80 ->bu kodu girdikten sonra shell’in açık kalması gerekiyor, kapatırsan bağlantı da gider.

==> VOLUME <==

-ephemeral volumes(geçici volumeler)
1-emptyDir -> pod bir node’a atandığında oluşturulur ve pod atandığı node’da çalıştığı sürece var olur. Başlangıçta boştur.
- Pod içindeki tüm containerlar emptyDir deki dosyaları okuyabiilir. bu boş klasör container içerisindeki herhangi bir path’ e mount edilebilir.
****bir pod herhangi bir nedenle bir node’dan silindiğinde emptyDir içindeki veriler de kalıcı olarak silinir**.(bu yüzden ephemeral volume denir)

2-hostPath -> worker node üzerindeki specific bir klasörün path’ini belirterek pod’a bağlamaya imkan verir. (kısıtlı bir kullanımı var, ne yaptığınızdan emin değilseniz uzak durun)

==> SECRET <==

-secret; pod-deployment gibi bir k8s objesidir.
-gizli bilgileri, pod-yaml dosyalarına veya container imajına koymak yerine Secret objesi içinde saklamak daha güvenli ve esnektir.

****oluşturulan secret’lar atanacak Pod’lar ile aynı namespace’de olmalı**

apiVersion: v1
kind: Secret

```
metadata:
  name: mysecret
type: Opaque
stringData:
  db_server: db.example.com
  db_username: admin
  db_password: P@ssw0rd!
```

type:Opaque-diğer ismi Generic(defaul-Opaque)
8 type secret vardır.->diğer tipler az kullanılır.
*stringData: ve Data kısımda secretlar yazılır.
Data kullanırsan Base64 ile encoded halini girmemiz gerekir.

-Imperative olarak secret oluşturma
$ kubectl create secret generic mysecret2 --from-literal=db_server=db.example.com --from-literal=db_username=admin --from-literal db_password=P@assw0rd!

**hassas bilgileri shell'e yazmamak için bunları başka txt dosyalarına yazıp aşağıdaki komutu kullanabiliriz.
$ kubectl create secret generic mysecret3 --from-file=db_server=server.txt --from-file=db_username=username.txt

-->Secret'lara Pod'a atama:
1-volume kullanarak
2-environment variable olarak

**bir secret oluşturduğumuz zaman bunlar ilk olarak etcd'de base64 encoded olarak tutuluyor(yani encrypted değil), cloud hizmet sağlayıcaları bunu encrypt eden hizmeti zaten otomatik sağlıyorlar, kendi cluster'ımızda ise kubernetes docs'dan encryptin data konusuna göz atmalıyız.

==>> CONFIG MAP

ConfigMap -> gizli olmayan verileri key/value değer eşlenikleri olarak depolamak için kullanılan objedir.
-Podlar, ConfigMap'i environment variable, komut satırı argümanı veya volume olarak bağlanan yapılandırma dosyaları olarak kullanılabilir.
*ConfigMap->Base64 encode edilmez,

**configmap' de type tanımlanmasına gerek yok.

$ kubectl create configmap myconfigmap --from-literal=background=blue --from-file=a.txt

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  db_server: "db.example.com"
  database: "mydatabase"
  site.settings: |
    color=blue
    padding:25px
```

=> Node affinity

-node affinity: node'lara atanan etiketlere göre podunuzun hangi node üstünde schedule edilmeye uygun olduğunu kısıtlamamıza olanak tanır.

```
---------------------------------------------
  affinity:
    nodeAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: app
            operator: In #In, NotIn, Exists, DoesNotExist
            values:
            - blue
---------------------------------------------
  affinity:
    nodeAffinity:
preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: app
            operator: In
            values:
            - blue
```

---
operatör çeşitleri:
#In, NotIn, Exists, DoesNotExist

weight--> 1-100 arasında seçilebiliyor, hangi tanımın daha öncelikli olarak değerlendirileceğini belirlemek için kullanılır.

***required'da uygun node bulunamazsa pod pending de bekler*
****preferred de tanıma en uygun node'da pod oluşturulur. Pending'e düşmez.*

=> Node'a label ekleme komutu
$ kubectl label node minikube(nodename) app=blue

==> Pod affinity

-pod affinity; podunuz hangi node üstünde oluşturulmaya uygun olduğunu, node'lardaki etiketlere göre değil, hali hazırda node üzerinde çalışmakta olan pod'lardaki etiketlere göre sınırlamanıza olanak tanır.

```
---------------------------------
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - frontend
        topologyKey: kubernetes.io/hostname
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              ...
---------------------------------
```

**** node'lar üzerinde önceden tanımlanmış bir çok label mevcuttur.
kubernetes.io/arch

kubernetes.io/hostname
kubernetes.io/os

**cloudprovider'ların yönetilen kubernetes hizmetlerindeki node'larda, o node'un hangi regionda bulunduğu aşağıdaki anahtarlar ile belirtili.
topology.kubernetes.io/region=northeurope
topology.kubernetes.io/zone=northeurope-1

****podAntiAffinity -> podlarda antiaffinity node'lardan farklı yapılır. podlarda ayrı olarak podAntiAffinity anahtarıyla ayrı bir tanım yapılması gerekir.

==>> TAINT and TOLERATION

*affinity tanımları bir pod'un nerede schedule edileceğini belirtir. Yani Pod'a göre bir tanımlamadır. Eğer uygulamaya göre bir tanım yapmak istiyorsak (şu worker node'da şu tipte bir uygulama çalışsın gibi) affinity kullanamayız.

Node'lara taint ekleme.
$ kubectl taint node "node_ismi" "anahtar=değer:eylem"
Ör: **kubectl taint node minikube platform=production:NoSchedule**
***
Node'lardan taint kaldırma.
$ kubectl taint node "node_ismi" "anahtar-"
Ör: **kubectl taint node minikube platform-**

```
  tolerations:
  - key: "platform"
    operator: "Equal"
    value: "production"
    effect: "NoSchedule"
```

```
  tolerations:
  - key: "platform"
    operator: "Exists"
    effect: "NoSchedule"
```

->bu pod şurada oluşturulsun ->> nodeAffinity
->worker node'un üzerinde sadece şu podlar çalışabilsin ->> taint-

toleration

==>DAEMONSET

-Deployment'a benzeyen bir kubernetes objesidir.
-DaemonSet, tüm/bazı node'ların bir Pod'un bir kopyasını çalıştırmasını sağlar.
-Her node'da çalışmasını isteyebileceğimiz log toplama, storage provisioning gibi uygulamaların kolay deploy edilmesini sağlar.
-her yeni eklenen node için tekrar tekrar ayar yapmaya gerek kalmaz.

**Deployment objesindeki gibi DaemonSet'de de label-selector tanımı vardır. DaemonSet oluşturacağı Pod'ları bu selector'e göre seçer. Bu nedenle aynı label tanımının spec'de de olması gerekir.

***Uygulamada master node'lar üzerinde Pod çalıştırılmaz, master node'larda **node-role.kubernetes.io/master.NoSchedule** şeklinde bir taint eklidir. Dolayısıyla bunu tolere edebilecek bir tanım ekli değilse Pod master node üzerinde schedule edilmez.

$ kubectl get daemonset
$ kubectl delete daemonset mydaemonset

->her node üstünde çalışması gereken bir tane pod oluşturmamız gerekiyorsa daemonSet kullanırız.

==> PERSISTENT VOLUME
==>PERSISTENT VOLUME CLAIM

CSI(ContainerStorageInterface); Kubernetes'in storage altyapısının nasıl planlaması gerektiğini belirten bir standart. Depolama çözümü üretenler bu standartlara uygun driver yazılımı yazarak kubernetes'in kendi altyapılarıyla konuşmasını sağlayabilir.

PersistentVolume
accessModes:
ReadWriteOnce - aynı anda tek bir pod'a bağlanabilir. bu pod bu volume hem yazar hem okur.
ReadOnlyMany - sadece okuma, birden fazla pod
ReadWriteMany - okuma-yazma, birden fazla pod

persistentVolumeReclaimPolicy

-Pod'un volume ile işi bittikten sonra volume'e nasıl davranılması gerektiğini belirlediğimiz kısım.
-retain ->olduğu gibi kalsın
-recycle -> volume kalıyor içindeki dosyalar siliniyor.
-delete -> volume tamamen siliniyor.

PersistentVolumeClaim
-bir persistenVolume'u bir pod'a bağlamak için önce PersistenVolumeClaim isminde bir k8s objesi yaratmak gerekiyor.

PVC-> sistemdeki PV'ler arasından işimize uygun olanı talep etmemize imkan verir.
**PVC selector kısmında PVC ile PV label'lar üzerinden eşleştirilir.

1-PV'ler k8s cluster'ı yöneten sistem yöneticileri veya adminler tarafından oluşturulur.
2-Developerlar ihtiyaçlarına göre PVC oluşturur.
3-PVC, clusterda mevcut ve en uygun PV ile eşleştirilir.
4-Pod tanımında volume anahtarı altında PVC ile Pod'u ilişkilendirilir ve ilgili container altında volumeMounts altında bağlanma pathi belirlenir.
5-Artık pod silinse bile yeni pod bu volume'u kullanır.

NFS Server
$ docker volume create nfsvol

$ docker network create --driver=bridge --subnet=10.255.255.0/24 --ip-range=10.255.255.0/24 --gateway=10.255.255.10 nfsnet

$ docker run -dit --privileged --restart unless-stopped -e SHARED_DIRECTORY=/data -v nfsvol:/data --network nfsnet -p 2049:2049 --name nfssrv ozgurozturknet/nfs:latest
---
***
Persistent Volume objelerinin listelenmesi
$ kubectl get pv
***
Persistent Volume objelerinin silinmesi
$ kubectl delete pv "pv_ismi
Ör: kubectl delete pv mypv
***
Persistent Volume Claim objelerinin listelenmesi

$ kubectl get pvc
***
Persistent Volume objelerinin silinmesi
$ kubectl delete pvc "pvc_ismi

Ör: kubectl delete pvc mypvc

****PV 'nin accessModes ile PVC'nin accessModes'u mutlaka eşleşmek zorunda**.

***PVC spec.selector.matchLabels kısmı ile hangi PV 'yi talep ettiğimi belirtmiş oluyorum.

-PVC'yi oluşturduktan sonra status kısmında Bound yazıyorsa PV ile PVC bağlandı/bind oldu demektir.

==> STORAGE CLASS <==

-CloudService sağlayıcılar ile kullanılır.
-PersistentVolume'u manuel oluşturmak yerine PVC'ye göre dinamik olarak oluşturabilme imkanı sağlar.
*Storage class bir template, bu template'ı kullanarak otomatik olarak PVC 'deki talebe göre oluşur.
-Storage class'da recyle seçeneği yok.
*VolumeBindingMode
-Immediate: hemen pvc'ye bind edilir.
-WaitForFirstCustormer: pod oluşturulup atanana kadar volume'un oluşturulmamasını öngörür.

NAME PROVISIONER
gp2 (default) kubernetes.io/aws-ebs

==>StatefulSet
*Deployment objesine benzer, fakat 3 temel farklılık vardır.
1-statefulSet ile oluşturulan her pod'un kendine ait bir pv'si olur.
2-Pod'lar sırayla oluşturulup, sırayla silinir. En son yaratılan pod ilk silinir.
3-İsimler random seçilmez, statafulSetName-0 1 şeklinde devam eden sabit bir isim alır.

==>JOB

-Job objesi, belirtilen sayda pod/uygulama görevini başarılı olarak tamamlayana kadar pod oluşturur, belirtilen sayıya ulaştığında job tamamlanır ve yeni pod oluşturulmaz.
-İş bittiğinde podlar silinmez, böylece logları kontrol imkanı vardır.
-Job silindiğinde, oluşturduğu pod'lar da silinir.

Kullanım Yerleri:
1-Tek seferlik çalışıp yapması gerekeni yapıp kapanan uygulamalarda,
2-Queue veya bucket' da işlenmesi gereken bir çok işimiz olduğunda bunları eritmek adına, bu işler eriyene kadar çalışacak uygulamalar job şeklinde deploy edilir.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 2
  completions: 10
  backoffLimit: 5
  activeDeadlineSeconds: 100
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never #OnFailure
```

==>CronJob
-Belirli bir zamanlamaya göre Cron formatında yazılmış bir Job'u periyodik olarak çalıştırmayı sağlayan objedir.

```
apiVersion: batch/v1beta1 # not stable until kubernetes 1.21.
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
```

Authorization
RBAC-> Role Based Access Control
rol tabanlı erişim denetimi
RBAC OBJELERİ
role
role binding
cluster role
cluster role binding
****Role ile ClusterRole arasındaki farklar:**
1-clusterRole'de belirlenen yetkiler tüm cluster çapında geçerlidir. Bu nedenle yaml dosyasında namespace tanımı yapılmaz.
2-Non-namespace objelerin oluşturulmasında ClusterRole kullanılır.

role --> role binding --> user

**ClusterRolleri, clusterRoleBinding yerine sadece RoleBinding ile birlikte kullanıp sadece namespace seviyesinde atama yapabiliriz istersek.

edit-> tüm kaynaklarda düzenleme yetkisi veren cluster role
view-> tüm kaynaklarda okuma hakkı verir.

=>>Service Account
-Uygulamaların authenticate olması için, podlarda çalışan processler tarafından kullanılmak üzere tasarlanmıştır.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: testsa
  namespace: default
```

*k8s her namespace için default bir serviceAccount oluşturur.
*ve her poda aksi belirtilmedikçe bu serviceAccount bağlanır.
*default service accountun hiçbir yetkisi yoktur.
*istenirse service accounta role bind edilebilir.

===>>> INGRESS <<<===
-Bölüm:64
-loadbalancer service
-path-based routing

->Ingress Controller
*Layer7 Application Loadbalancer kavramınının k8s özelliklerine göre çalışan ve k8s'e deploy ederek kullanabildiğimiz bir türü.
(loadbalancerın k8s'e özgü şekli)-Nginx, HAproxy, Traefik
-ıngress controller'ı kubernetes cluster'ımıza kurduğumuzda bu controller dış dünyaya bir loadbalancer ile expose ediliyor ve bir IP adresine sahip oluyor.
*Dış dünyadan kubernetes'e erişim artık sadece bu IP adresinden yapılabiliyor.
*hangi service hangi URL ile ulaşılacağını(IngressController configuration) ingress objeleri aracılığıyla belirliyorus

INGRESS OBJECTS
***Ingress,** genellikle HTTP olmak üzere bir clusterdaki servislere dışardan erişimi yöneten bir API objesidir.
-ingress controller ayarlarını bu objelerle yapıyoruz.
-ingress controller layer 7 appLoadbalancer'larını sunduğu SSL termination, path-based routing gibi özellikleri deploy edebilmemizi sağlar.

-En çok kullanılan ingress controller'lar
1-nginx 2-Traefik

**ingress kurduğumuz zaman nginx kendisine ingress-nginx diye bir namespace oluşturur ve gerekli şeyleri buraya kurar, bunları görmek için
$ kubectl get namespoces
$ kubectl get all ---> deploy edilen tüm objeleri gösterir.
$ kubectl get all -n ingress-nginx

--bir kere ingress kurduktan sonra artık cluster'ın giriş noktası bu ingress olur.

**annotations --> cluster'a deploy edilen ingress controller üzerinde herhangi bir ayar yapılmak istenirse annotations ile yapılır. mesela, sertifika bilgilerinin girilmesi.

***bir cluster da birden fazla ingress oluşturulabilir, tek oluşturma zorunlu değil. Zaten tüm veriler, ayarlar ingress controller da toplanıyor.

Configure RBAC in Kubernetes Like a Boss
Emre Savcı -Medium

->important concepts of RBAC
1-subjects; users, groups, service accounts
2-resources: k8s API objects
3-verbs: operations

*if you need define permissions inside a namespace use Role, if you need that cluster wide use ClusterRole

$ kubectl create role my-custom-role --verb=get --verb=list --resource=pods --resource=services --k8boos

cordon -> node'a yeni pod schedule edilmesini istemediğimizde kullanırız, schedule trafiğine tekrar açmak istersek uncordon komutu kullanılır.

drain -> node içindeki podların başka node'lara tahliye edilmesini istediğimizde kullanırız, drain içinde cordon komutununu kapsar

***port-forward**
$ kubectl port-forward deployment/deployname -n test 8080:80

->kubernetes dashboard-> kubernetesin resmi gui'si
-> lens, headlamp -3rd part gui'ler

****static pod:**
-kubelet içinde belirlediğimiz folder'a yaml dosyalarını koyarak apiserver kullanmadan pod oluşturmasını isteyebiliriz, az kullanılsa da bilmek gerekir.
kube-manifest-path etc/kubernetes/manifests/

****Network Policy**
->NetworkPolicy oluşturulmadıysa;
-cluster içindeki her pod diğer her pod ile,
-her pod dış dünya ile(node'un route ayarları var ise),
-herkes dış dünyadan pod ile(ingress veya load balancer ile açılmış ise),

konuşabilir.
->bu durumu kısıtlamak istersek network policy kullanılır.

=>> HELM
-Kubernetes package manager

*chart-> kubernetes üzerinde yükleyebildiğimiz uygulamanın paketlenmiş hali, a chart is a Helm package, it contains all of the resource definitions necessary

*release-> is an instance of a chart running in a kubernetes cluster. One chart can often be installed many times into the same cluster. And each time it is installed, a new release is created. Chart'ın kubernetesin üzerine yüklenmiş haline denir.

*repository -> helm chartların tutulduğu yerler, A repository is the place where charts can be collected and shared.

*Helm installs chart into Kubernetes, creating a new release for each installation.

-bir repository oluşturuluyor ve helm chartlar buraya ekleniyor, bu chartlardan birini ben kendi cluster'ımda kurmak istiyorsam öncelikle bu repoyu kendi hostumdaki helm'ime yüklüyorum, yani helm'in arama yapacağı listelere eklemiş oluyorum, daha sonra bu chartları kendi cluster'ıma kurabiliyorum. Kurduğum zamanda bunun adı release oluyor.

*ArtifactHub -> helm chart'ları arayabildiğiz site, dockerhub gibi
$ helm search hub wordpress
$ helm search repo wordpress -> kendi bilgisayarımdaki helm listesine ekledim repository'lerde arama yapmak için.
helm repo add bitnami https://charts.bitnami.com/bitnami
$ helm install my-release bitnami/wordpress
$ helm status
$ helm show values bitnami/wordpress
$ helm upgrade
$ helm rollback

==>> PROMETHEUS

-K8s'te izlenmesi gereken 4 farklı yer/durum vardır.
1-Kubernetes Cluster'ın durumu
-Cluster o anda nasıl çalışıyor, hangi objeler var, neler deploy edilmiş.
2-Objelerin mevcut durumu nedir. Deployment'a uygun olarak objeler çalışıyor mu, node ne kadarlık cpu tüketiyor
3-Node'ların izlenmesi gerekir, mevcut cpu, memory, i/o kullanımı nasıl, network trafiği ne kadar
4-Uygulamalar nasıl çalışıyor, logların takibi...

->k8s'de bilgi almamıza yarayacak komutlar
kubectl get
kubectl describe
kubectl top
kubectl logs

*prometheus kubernetes üzerinde monitoring'i otomatikleştiren bir uygulamadır.
-takibi gereken durumlardan ilk üçünü prometheus ile yapabiliriz.

->prometheus bir metrics altyapısı/sunucusudur. (cnsf projesidir)
-en büyük özelliği pull-base çalışması, yani metrikleri otomatik topluyor, agent kullanmıyor.

*kubernetes için en çok kullanılan metrics toplama aracı haline geldi.

**normalde prometheus tüm gerekliliklerıyle install etmek karmaşık bir işlem, fakat prometheus-community adlı oluşum bu işlemi helm chart'lar aracılığıyla basitleştirmiş.
=>Prometheus Kubernetes Stack
github.com/prometheus-community/helm-charts

=>EFK
-Elasticsearch Fluentd Kibana stack
-kibana

*log'ları toplamak ve monitor etmek için kullanılır.
-elastik search aslında bir search engine, kibana bunu görselleştirmeye yarıyor.

**prometheus-grafana metriclerle, elastic ve kibana loglarla ilgili.

--logları logstash veya fluentd ile araçları ile topluyoruz.
-fluentd->kubnernetes cncf projesi