Özgür ÖZTÜRK Hocamızın,
https://www.udemy.com/course/kubernetes-temelleri/
eğitimini esas alarak çeşitli kaynaklarla zenginleştirdiğim

KUBERNETES

Notları

2023

Kazim Esen

Faydalanılan kaynaklar;
https://berksafran.gitbook.io/kubernetes-notlari/
https://kubernetes.io/
https://medium.com/devopsturkiye/kubernetes-nedir-eb5c5d149e69

KUBERNETES - k8s

DevOps süreçlerini otomatize ederek kolaylaştıran araçlara "Container Orchestration" araçları denir.

Kubernetes; tüm sistem konfigürasyonlarını ayarlayıp, bu karar mekanizmalarını yöneten en yaygını kullanılan Container Orchestration'dır.
Kubernetes hem beyan temelli yapılandırmayı hem de otomasyonu kolaylaştıran, container iş yüklerini ve hizmetleri yönetmek için oluşturulmuş, taşınabilir ve genişletilebilir açık kaynaklı bir platformdur. Kubernetes Google tarafından GO dilinde geliştirilmiş Cloud Native Computing Foundation(CNCF) tarafından desteklenen mevcut konteyner haline getirilmiş uygulamaları otomatik deploy etmek, sayılarını arttırıp azaltmak gibi işlemler ile birlikte yönetmeyi sağlayan bir konteyner kümeleme (container cluster) aracıdır.

Kubernetes Tasarımı ve Yaklaşımı

Birden fazla geliştirilebilir modüllerden oluşmaktadır. Bu modüllerin hepsinin bir görevi vardır ve tüm modüller kendi görevlerine odaklanır. İhtiyaç halinde bu modüller veya yeni modüller geliştirilebilir. (extendable)

K8s; "Şunu yap, sonra şunu yap" gibi adım adım ne yapılacağını söylemek yerine (imperative yöntem), "Şunu istiyorum" (declarative yöntem) yaklaşımı sunmaktadır.
Nasıl yapılacağını tarif etmek yerine, ne istendiği söylenir.
Imperative yöntem, zaman kaybettirir, tüm adımlar tasarlanmak zorundadır.
Declarative yöntem'de ise sadece ne istendiği söylenerek sonuca bakılıyor.
Kubernetes ne istendiğini bir yaml dosyasından öğreniyor ve bu dosyada istenenlerin dışına çıkmıyor.
Desired State, Deklara Edilen Durum yani istekler gibi düşünülebilir.

Kubernetes Nasıl Çalışır?

1. Kubectl(kubernetes client) isteği API server'a iletir.
2. API Server isteği kontrol eder etcd'ye yazar.
3. etcd yazdığına dair bilgilendirmeyi API Server'a iletir.
4. API Server, yeni pod yaratılacağına dair isteği Scheduler'a iletir.
5. Scheduler, pod'un hangi server'da çalışacağına karar verir ve bunun bilgisini API Server'a iletir.
6. API Server bunu etcd'ye yazar.
7. etcd yazdığına dair bilgiyi API Server'a iletir.
8. API Server ilgili node'daki kubelet'i bilgilendirir.
9. Kubelet, Docker servisi ile docker soketi üzerindeki API'yi kullanarak konuşur ve konteyner'ı yaratır.
10. Kubelet, pod'un yaratıldığını ve pod durumunu API Server'a iletir.
11. API Server pod'un yeni durumunu etcd'ye yazar.

Tercih edilen OS'ne göre; **`kubectl`**, çeşitli Linux platformları, macOS ve Windows üzerine:

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

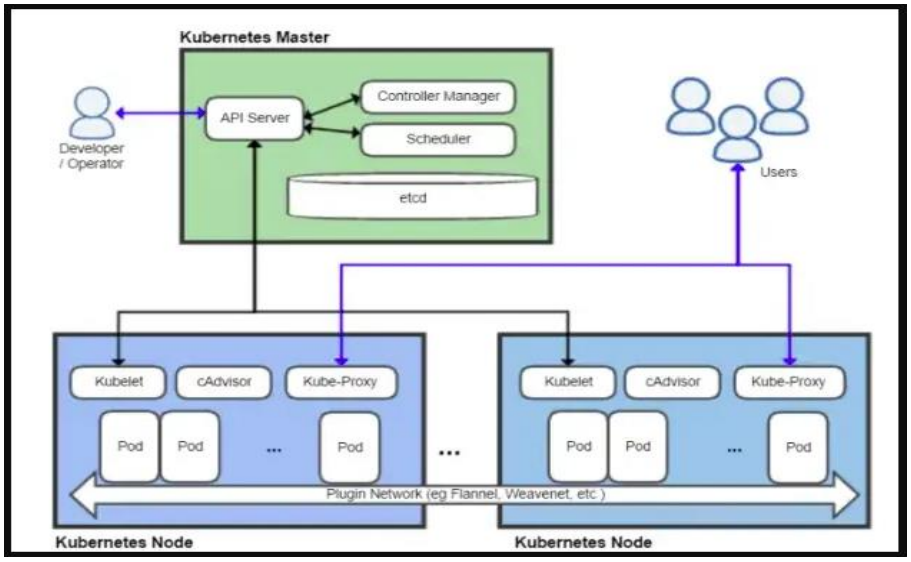
linkinden kurulabilir.

Kubernetes Componentleri

K8s, microservice mimarisi dikkate alınarak oluşturulmuştur.

Control Plane (Master Nodes)

Master-node: Yönetim modulleri API Server, etcd, Controller Manager ve Scheduler'in çalıştığı yerdir.
Worker-node: İş yükünün çalıştığı yerdir.



kube-apiserver (api): Master sunucuya gelen tüm REST requestlerin yönetilmesinden sorumludur. Cluster'ın beyni, ana haberleşme merkezi, giriş noktasıdır. Tüm componentler ve node'lar, kube-apiserver üzerinden iletişim kurar. Ayrıca, dış dünya ile platform arasındaki iletişimi de kube-apiserver sağlar. Json file'ları ile yönetilir. Authentication ve Authorization görevini üstlenir.

etcd: Tüm cluster verisi, metada bilgileri ve kubernetes platformunda oluşturulan componentlerin ve objelerin bilgileri, key-value biçiminde datalarını tutar. Diğer componentler, etdc ile direkt haberleşemezler ve iletişimi, kube-apiserver aracılığıyla yaparlar. Coreos tarafından yaratılmış open source distributed, tutarlı ve izlenebilir bir key value store (nosql database) 'dur.

kube-scheduler (sched): Çalışma planlamasını yapar. Yeni oluşturulan ya da bir node ataması yapılmamış Pod'ları izler ve üzerinde çalışacakları bir node seçer. (Pod = container) Bu seçimi yaparken, CPU, Ram vb. çeşitli parametreleri değerlendirir ve bir seçme algoritması sayesinde pod için en uygun node'un hangisi olduğuna karar verir. Bir pod'un hangi node üzerinde çalışacağına karar verir, kubelet'i tetikler ve ilgili pod ve içindeki konteyner oluşturulur. Kısacası yeni bir pod oluşturulması isteğine karşı API server'ı izler.

kube-controller-manager (c-m): K8s'in kontrollerinin yapıldığı yapıdır. Mevcut durum ile istenilen durum arasında fark olup olmadığını denetler. Temel olarak, bir denetleyicidir, kümenin durumunu API Server izleme özelliğiyle izler ve bildirildiğinde, geçerli durumu istenen duruma doğru hareket ettirmek için gerekli değişiklikleri yapar. Örneğin; 3 cluster istendi, k8s bunu gerçekleştirdi fakat bir sorun oldu ve 2 container kaldı. kube-controller, burada devreye girer ve hemen bir cluster daha ayağa kaldırır.
Tek bir binary olarak derlense de içerisinde bir çok controller barındırır: Node Controller, Job Controller, Service Account & Token Controller, Endpoints Controller.

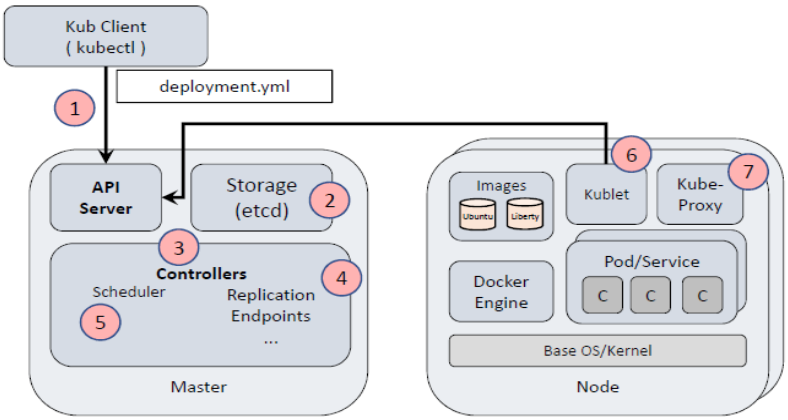
Worker Nodes

Container'ların çalıştığı yerlerdir. Container veya Docker gibi container'lar çalıştırır. Her worker node'da 3 temel component bulunur:

Container runtime: Containerları çalıştırmaktan sorumludur. Kubernetes container runtime olarak; Docker, containerd, CRI-O destekler. Varsayılanı Docker'dır. Ama çeşitli sebeplerden dolayı Docker'dan Containerd'ye geçmiştir. Docker ve containerd arasında fark yok nedebilecek kadar azdır. Hatta Docker kendi içerisinde de containerd kullanır. Konteyner yönetimini yapar. İmage'ı ilgili repo'dan çeker ve konteyner'ların start ve stop olmalarını sağlar. Container Engine olarakta adlandırılır.

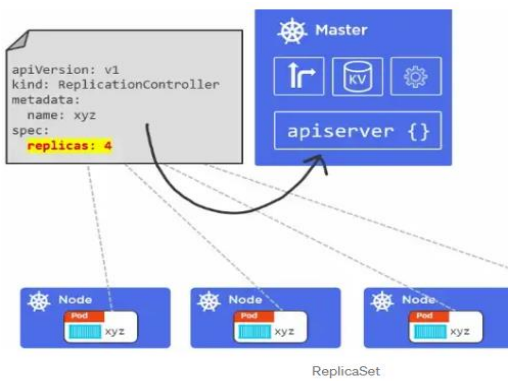
kubelet: API Server aracılığıyla etcd'yi kontrol eder ve sched tarafından bulunduğu node üzerinde çalışması gereken podları yaratır. Containerd'ye haber gönderir ve belirlenen özelliklerde bir container çalışmasını sağlar. Node üzerindeki ana kubernetes ajanıdır. API server'dan gelecek isteklere karşı API Server'ı izler. İlgili docker servisi ile konuşarak Pod'u ayağa kaldırır ve bunun bilgisini API Server'a iletir.
Cluster'daki her node çalışan bir agent'tır. Pod içerisinde tanımlanan containerların çalıştırılmasını sağlar. Çeşitli mekanizmalar aracılığıyla sağlanan bir dizi Pod tanımı alır ve bu Pod tanımında belirtilen containerların çalışır durumda ve sağlıklı olmasını sağlar.

kube-proxy: Nodelar üstünde ağ kurallarını ve trafik akışını yönetir. Bu ağ kuralları, cluster'in içindeki veya dışındaki ağ oturumlarından Pod'larla ağ iletişimine izin verir, izler. Kubernetes network'u diyebiliriz. Pod'lara IP adresi proxy sayesinde atanır. Bir pod'un içindeki tüm konteyner'lar bir adet paylaşımlı IP'yi kullanır. Kube-proxy aynı zamanda bir servisin altındaki tüm pod'lara load-balance özelliği kazandırır.



1. Kullanıcı "**kubectl**" aracılığıyla yeni bir uygulama kuruyor
2. API server isteği alır ve DB'de saklar (**etcd**)
3. İzleyiciler/kontrolörler kaynak değişikliklerini algılar ve buna göre hareket eder
4. ReplicaSet izleyici/denetleyici yeni uygulamayı algılar ve istenen sayıda örnekle eşleşen yeni Pod('lar) oluşturur
5. Scheduler (Zamanlayıcı), bir kubelet'e yeni Pod('lar) atar
6. Kubelet, Pod'ları algılar ve çalışan container aracılığıyla dağıtır (ör. Docker)
7. Kubeproxy, hizmet keşfi ve yük dengeleme dahil olmak üzere Pod'lar için ağ trafiğini yönetir

ReplicaSet: Herhangi bir zamanda bir poddan kaç tane çalışacağı replicaset ile belirtilir. İstendiği anda bir pod kesintisiz bir şekilde scale edebilir. Genellikle belirli sayıda özdeş Pod'un kullanılabilirliğini garanti etmek için kullanılır.
Namespace: Namespace ortamları birbirinden izole eder.
Service: Pod'ların ön tarafında konumlanan ve gelen istekleri karşılayıp arka tarafa pod'lara gönderen katmandır. Blue-green deployment, pod scaling gibi işlemlerin kesintisiz yapılmasını sağlar.
Secret: Parola, kullanıcı, token gibi bilgileri güvenli bir şekilde depolayarak, uygulamaların güvenli yönetilmesini sağlar.



Komut satırı aracı (kubectl)

Kubernetes, kubernetes API kullanarak bir kubernetes kümesinin kontrol düzlemiyle iletişim kurmak için kubectl komut satırı aracını sağlar.

Konfigürasyon için kubectl, $HOME/.kube dizininde config adlı bir dosya arar. KUBECONFIG ortam değişkenini veya --kubeconfig bayrağını ayarlayarak farklı bir kubeconfig dosyası belirlenebilir.

Docker komut satırı aracındaki kubectl, kubernetes için bazı eşdeğer komutları açıklar.

kubectl config dosyası

-kubectl aracı bağlanacağı Kubernetes cluster bilgilerine config dosyaları aracılığıyla erişir.
-Config dosyasında Kubernetes cluster bağlantı bilgileri ve bağlanırken kullanılacak kullanıcılar belirtilir.
-Daha sonra bu bağlantı bilgileri ve kullanıcıları ve ek olarak namespace bilgilerini de oluşturarak context'ler yaratılır.
-Kubectl varsayılan olarak $HOME/.kube/ altındaki config dosyasına bakar ama bu KUBECONFIG enviroment variable değeri değiştirilerek güncellenebilir.

kubectl ile mevcut cluster'ın yönetimi config dosyası üzerinden yapılması gerekir. Minikube gibi tool'lar config dosyalarını otomatik olarak oluşturur, default config dosyası **~/.kube/config** 'dir.

context: Cluster ile user bilgilerini birleştirerek context bilgisini oluşturur.
"Bu cluster'a bu user ile bağlanacağım." anlamına geliyor.

Kubectl Config Komutları

`kubectl config` Ayarları düzenler veya gösterir.
`kubectl config get-contexts` mevcut contextleri listeler.
* işareti kullanılan context'i gösterir, yapılan tüm işlemler bu context'te gerçekleşir.
`kubectl config current-context` current context'i verir.
`kubectl config use-context <contextName>` Current context'i contextName ile değiştirir.
`kubectl config use-context docker-desktop` docker-desktop context'ine geçer.

kubectl Komut Parametreleri

kubectl komutlarının şematik bir tasarımı vardır:

```
kubectl <fiil> <object>
 # <fiil> = get, delete, edit, apply
 # <object> = pod
```

kubectl'de aksi belirtilmedikçe tüm komutlar config'deki default namespace'de uygulanır.

`kubectl cluster-info` Cluster ile ilgili bilgileri gösterir.
`kubectl get pods` Default namespace'deki pod'ları getirir.
`kubectl get pods testpod` testpod isimli pod bilgilerini getirir.
`kubectl get pods -n kube-system` kube-system namespace'indeki pod'ları getirir.
`kubectl get pods -A` Tüm namespacelerdeki pod'ları getirir.
`kubectl get pods -A -o <wide|yaml|json>` Pod'ları istenilen formatda getirir.

```
# brew install jq                          //komutu ile json query plugini kurulduktan sonra;
kubectl get pods -A -o json | jq -r ".items[].spec.containers[].name"
```

`kubectl apply --help` apply fiilinin kullanımı ile ilgili bilgi verir, bir obje ile ilgili bilgi vermez.
`kubectl explain pod` pod objesinin ne olduğunu, hangi field'ları aldığını gösterir.
`kubectl get pods-w` kubectl'i izleme (watch) moduna alır ve değişimler canlı olarak izlenir.
`kubectl get all -A` Çalışan tüm objelerin durumunu gösterir.
`kubectl exec -it <podName> -c <containerName> -- bash`
Pod içerisinde çalışan bir container'a bash ile bağlanır.

Pod

K8s üzerinde koşan, çalışan, deploy edilen şeylere Kubernetes Object denir. En temel object olan Pod, Kubernetes'te oluşturabileceğiniz ve yönetebileceğiniz en küçük birimleridir.
Konteyner'ların çalışma alanı denebilir. Bir pod bir veya birden fazla konteyner barındırabilir ama Best Practice olarak her bir pod bir tek container barındırır. Kubernetes tarafında çalışma şekli gereği yeni bir deploy isteği geldiğinde pod'un yeni versiyonu oluşturulur ve çalıştığı görüldüğünde diğer pod versiyonu kapatılır. Dolayısıyla pod içerisinde birden fazla konteyner olduğu durumda diğer uygulamala konteyner'ları etkileneceğinden bir pod içerisinde bir konteyner tavsiye edilir. Pod öldüğünde tekrar geri kalkmaz aynı imajdan onun yerine yeni bir pod ortaya çıkar. Her pod'un uniq bir id'si ""uid"" ve unique bir IP'si vardır. Api-server, bu uid ve IP'yi etcd'ye kaydeder. Scheduler ise herhangi bir podun node ile ilişki kurmadığını görürse, o podu çalıştırması için uygun bir worker node seçer ve bu bilgiyi pod tanımına ekler. Pod içerisinde çalışan kubelet servisi bu pod tanımını görür ve ilgili container'ı çalıştırır. Aynı pod içerisindeki containerlar aynı node üstünde çalışıtırılır ve bu containerlar birbirleriyle localhost üstünden haberleşirler.

`kubectl run firstpod --image=nginx --restart=Never --port=80 --labels="app=frontend"`
Pod oluşturur ve **`--restart`** alt parametresine "eğer pod içerisindeki container herhangi bir nedenle durursa, tekrar çalıştırılmaması için" **`Never`** ataması yapılır.

`kubectl get pods -o wide` Pod'ları Listeler.

`kubectl describe pods first-pod` Object Detayları gösterilir.
Bu bilgilerden **Events**; Pod'un tarihçesini, neler olduğunu, k8s'in neler yaptığını gösterir.
- Önce Scheduler node atamış,
- kubelet container image'ı pull etmiş,
- kubelet pod'u oluşturmuş.

`kubectl logs <podName>` Pod Loglarını gösterir.
`kubectl logs -f <podName>` Logları Realtime olarak gösterir.

`kubectl exec <podName> -- <command>` Komut çalıştırır.
`kubectl exec first-pod -- ls /"`

`kubectl exec -it <podName> -- <shellName>` Container'a bağlantı sağlar.
`kubectl exec -it first-pod -- /bin/sh"`
Bir pod içerisinde birden fazla container varsa;
`kubectl exec -it <podName> -c <containerName> -- <bash|/bin/sh>`

`kubectl delete pods <podName>` Pod'u siler.

YAML

k8s, declarative yöntemi uygularken YAML veya JSON dosya formatını kullanır.
--- (üç tire) ile bir YAML dosyası içerisinde birden fazla object belirtilebilir.
apiVersion:
kind:
metadata:
spec:
Her türlü object oluşturulurken; apiVersion, kind ve metadata olmak zorundadır.
kind –> Oluşturulacak object türü belirtilir. ÖR: pod

apiVersion –> Object'in hangi API üzerinde ya da endpoint üzerinde sunulduğunu gösterir.
Dokümantasyona bakarak veya kubectl aracı ile öğrenilir.
kubectl explain pods komutu ile pod'un özellikleri öğrenilebilir.
Komut çıktısındaki; **Versions** değeri **apiVersion**'dur.

metadata –> Object ile ilgili unique bilgiler tanımlanır. ÖR: namespace, annotation vb.
spec –> Oluşturulacak object'in özellikleri belirtilir. Her object için girilecek bilgiler farklıdır.

```
apiVersion: v1
kind: Pod
metadata:
    name: first-pod            # Pod'un ismi
    labels:                    # Atayacağımız etiketleri yazabiliriz.
        app: front-end         # app = front-end label'i oluşturduk.
spec:
    containers:                # Container tanımlamaları yapıyoruz.
    - name: nginx              # Container ismi
        image: nginx:latest
        ports:
        - containerPort: 80    # Container'e dışarıdan erişilecek port
```

K8s'e YAML Dosyasını Declare Etme

kubectl apply -f pod.yaml komutu ile **pod.yaml** dosyasında tanımlanan objeler oluşturulur.
kubectl describe pods firstpod ile oluşturulan pod'un tüm özellikleri görüntülenir.
YAML yöntemi kullanmak pipeline'da yaratmakta kullanılabilir.
Declarative Yöntem Avantajı –> Imperative yöntemle bir pod tanımladığında, bir özelliği update etmek istenirse "**Already exists**" hatası alınır. Ama declarative olarak YAML değiştirilerek **update** edilip, **apply** edilirse "**pod configured success**" mesajı alınır.

kubectl delete -f pod.yaml
YAML dosyası silme amacıyla yönlendirilirse, o dosyayla ilişkili kaynakları siler.

kubectl edit pods <podName>
Pod'u varsayılan editör ile düzenlemeyi ve değiştirmeyi sağlar. Tercih edilen bir yöntem değildir.

Pod Yaşam Döngüsü

- Pending –> Bir pod oluşturmak için YAML dosyasındaki ayarlar ile varsayılanlar harmanlanır ve etcd'ye kaydolur.
- Creating –> kube-sched, etcd'yi sürekli izler ve herhangi bir node'a atanmamış pod görürse devreye girerek; en uygun node'u seçer ve node bilgisini ekler.
Eğer bu aşamada takılı kalıyorsa, **uygun bir node bulunamadığı anlamına gelir.**
- **etcd**'yi sürekli izler ve bulunduğu node'a atanmış podlar varsa container'ları oluşturmak için image'leri download eder. Eğer image bulunamazsa veya repodan çekilemezse **ImagePullBackOff** durumuna geçer.
- Eğer image doğru bir şekilde çekilir ve container'lar oluşmaya başlarsa Pod **Running** durumuna geçer.

Burada "Pod Yaşam Döngüsü"ne bir ara verip, container'ların çalışma mantığından bahsedelim:
- Container image'larında sürekli çalışması gereken bir uygulama bulunur. Bu uygulama çalıştığı sürece container da çalışır durumdadır. Uygulama 3 şekilde çalışmasını sonlandırır:
  - Uygulama tüm görevlerini tamamlar ve hatasız kapanır.
  - Kullanıcı veya sistem kapanma sinyali gönderir ve hatasız kapanır.
  - Hata verir, çöker, kapanır.

"Pod Yaşam Döngüsü"ne geri dönelim:

- Container uygulamasının durmasına karşılık, Pod içerisinde bir **RestartPolicy** tanımlanır ve 3 değer alır:
  - **Always** -> Kubelet bu container'ı yeniden başlatır.
  - **Never** -> Kubelet bu container'ı yeniden başlatmaz.
  - **On-failure** -> Kubelet sadece container hata alınca başlatır.
- **Succeeded** -> Pod başarıyla oluşturulmuşsa bu duruma geçer.
- **Failed** -> Pod başarıyla oluşturulmamışsa bu duruma geçer.
- **Completed** -> Pod başarıyla oluşturulup, çalıştırılır ve hatasız kapanırsa bu duruma geçer.
- **CrashLookBackOff** -> Pod oluşturulup sık sık kapanıyorsa ve RestartPolicy'den dolayı sürekli yeniden başlatılmaya çalışılıyorsa, k8s bunu algılar ve podu bu state'e getirir.

Bu state de olan podlar incelenmelidir.

Multi Container Pods

Bir container'da iki uygulama çalıştırılmasının en önemli sakıncası; izolasyondur. İzolasyon sağlanmazsa, yatay scaling yapılamaz. Gerekli durumlarda 2. container yayına alındığında; 2 tane MySQL, 2 tane Wordpress olacak demektir ki bu iyi bir şey değildir. Bu sebeple 1 Pod = 1 Container = 1 uygulama olmalıdır!
Bazı uygulamalar bütünleşik (bağımlı) çalıştığından; pod'ların multi-container'a izin verir. Yani ana uygulama çalıştığında çalışmalı, durduğunda durmalıdır. Bu gibi durumlarda bir pod içerisine birden fazla container koyulabilir. Bir pod içerisindeki 2 container localhost üzerinden haberleşir.

Init Container İle Bir Pod İçerisinde Birden Fazla Container Çalıştırma

Go'daki init() komutu gibi ilk çalışan container'dır. Örneğin, uygulama container'ın başlayabilmesi için bazı config dosyalarını fetch etmesi gerekir. Bu işlemi init container içerisinde yapabiliriz.
1- Uygulama container'ı başlatılmadan önce **Init Container** ilk olarak çalışır.
2- Init Container yapması gerekenleri yapar ve kapanır.
3- Uygulama container'ı, Init Container kapandıktan sonra çalışmaya başlar.

Init Container kapanmadan uygulama container'ı başlamaz.

Label, Selector, Annotation

Label Nedir?

Service, deployment, pods gibi objeler arasında bağ kurmak için kullanılır.

kubernetes.io/ ve k8s.io/ kubernetes core bileşenler için ayrılmıştır, kullanılamazlar.

Tire, alt çizgi, noktalar içerebilir. Türkçe karakter kullanılamaz.

Label & Selector Uygulama

Label tanımı **metadata** tarafında yapılır. Aynı objeye birden fazla label eklenemez.

Label, gruplandırma ve tanımlama imkanı verir. CLI tarafında listelemekte kolaylaşır.

Selector - Label'lara göre Object Listelemek

Bir yaml dosyasından alınan bir kesit;

```
---
apiVersion: v1
kind: Pod
metadata:
  name: pod8
  labels:
    app: firstapp # burada app key, firstapp ise value'su.
    tier: backend # tier başka bir key, backend value'su.
...
---
```

"app" key'ine sahip objeleri listelemek için;

```
kubectl get pods -l <keyword> --show-labels
```

Equality based Syntax'i ile listeleme

```
kubectl get pods -l "app" --show-labels
kubectl get pods -l "app=firstapp" --show-labels
kubectl get pods -l "app=firstapp, tier=front-end" --show-labels
```

app key'i firstapp olan, tier'ı front-end olmayanlar:

```
kubectl get pods -l "app=firstapp, tier!=front-end" --show-labels
```

app anahtarı olan ve tier'ı front-end olan objectler:

```
kubectl get pods -l "app, tier=front-end" --show-labels
```

Set based ile Listeleme

app'i firstapp olan objectler:

```
kubectl get pods -l "app in (firstapp)" --show-labels
```

app'i sorgula ve içerisinde "firstapp" olmayanları getir:

```
kubectl get pods -l "app, app notin (firstapp)" --show-labels
kubectl get pods -l "app in (firstapp, secondapp)" --show-labels
```

app anahtarına sahip olmayanları listele

```
kubectl get pods -l "!app" --show-labels
```

app olarak firstapp atanmış, tier keyine frontend değeri atanmamışları getir:

```
kubectl get pods -l "app in (firstapp), tier notin (frontend)" --show-labels
```

Equality based ilk syntax ile bir sonuç bulunamazken, set based selector ikinci syntax ile sonuç bulunur:

```
kubectl get pods -l "app=firstapp, app=secondapp" --show-labels     # Sonuç yok!
kubectl get pods -l "app in (firstapp, secondapp)" --show-labels   # Sonuç var.
```

Komut ile label ekleme

```
kubectl label pods <podName> <label>
kubectl label pods pod1 app=front-end
```

Komut ile label silme Sonundaki **-** (tire) **Sil** anlamına gelir.

```
kubectl label pods pod1 app-
```

Komut ile label güncelleme

```
kubectl label --overwrite pods <podName> <label>
kubectl label --overwrite pods pod9 team=team3
```

Komut ile toplu label ekleme Tüm objelere bu label eklenir.

```
kubectl label pods --all foo=bar
```

Objeler Arasında Label İlişkisi

Normalde kube-sched kendi algoritmasına göre bir node seçimi yapar. Aşağıdaki gibi **hddtype** label'ına sahip node'u seçmesini sağlayarak, pod ile node arasında label'lar aracılığıyla bir ilişki kurmuş oluruz.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod11
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
  nodeSelector:
    hddtype: ssd
```

minikube cluster'ı içerisindeki tek node'a hddtype: ssd label'ı ekleyebiliriz. Bunu ekledikten sonra, pod "Pending" durumundan, aradığı node'u bulduğu için "Running" durumuna geçecektir.

```
kubectl label nodes minikube hddtype=ssd
```

Annotation

Aynı label gibi davranır ve metadata altına yazılır.
Label objeler arasında ilişki kurmak için kullanıldığından hassas bilgi sınıfına girer.
Annotation, label olarak kullanılmayacak önemli bilgileri kayıt altına almaya yarar.

```
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
```

Komut ile Annotation ekleme/silme

```
kubectl annotate pods annotationpod foo=bar     # Ekler.
kubectl annotate pods annotationpod foo-        # Siler.
```

Namespace

Kubernets'de namespaces, tek bir cluster içindeki kaynak gruplarını izole etmek için bir mekanizma sağlar. Namespaces'de bir k8s objesidir. Tanımlarken (özellikle YAML) dosyasında buna göre tanımlama yapılmalıdır. Namespaces'lerin birbirinden bağımsız ve benzersiz olması gerekir. Namespaces'ler birbiri içerisine yerleştirilemez. Her k8s oluşturulduğunda 4 default namespace oluşturulur:
`default, kube-node-lease, kube-public, kube-system`

Namespace Listeleme

Varsayılan olarak tüm işlemler ve objectler default namespace altında işlenir. **`kubectl get pods`** komutu herhangi bir namespace belirtilmediği için, default namespace altındaki podları getirir.

`kubectl get pods --namespace <namespaceName>` # -n ile --namespaces aynıdır.
`kubectl get pods -A` # -A veya --all-namespaces, tüm namespaces'lerdeki podları listeler.

Namespace Oluşturma

`kubectl create namespace <namespaceName>`
`kubectl get namespaces`

YAML dosyası kullanarak Namespace oluşturma

```
apiVersion: v1
kind: Namespace # namespace oluşturulmaya başlanıyor.
metadata:
  name: development # namespace isimlendiriliyor ve development ismi veriliyor.
---
apiVersion: v1
kind: Pod
metadata:
  namespace: development # oluşturulan namespace altında pod tanımlanıyor.
  name: namespacepod
spec:
  containers:
  - name: namespacecontainer
    image: nginx:latest
    ports:
    - containerPort: 80
```

Bir namespace içinde pod yaratırken, poda bağlanırken; kısacası podlar üzerinde herhangi bir işlem yaparken namespace belirtilmek zorundadır. Belirtilmezse, işlem default namespace altında yapılır.

Varsayılan Namespace'i Değiştirmek

`kubectl config set-context --current --namespace=<namespaceName>`

Namespace'i Silmek

DİKKAT! Namespace silerken confirmation istenmez. Namespace altındaki tüm objelerde silinir!
`kubectl delete namespaces <namespaceName>`

Deployment

K8s kültüründe "Singleton (Tekil) Pod"lar genellikle yaratılmaz. Bunları yöneten üst seviye objeler yaratılır ve bu podlar bu objectler tarafından yönetilir. (ÖR: Deployment)

Peki, neden yaratılmıyor?

Mesela, bir frontend objesinin bir pod içerisindeki container ile deploy edildiğini varsayalım. Eğer bu container'da bir hata meydana gelirse ve RestartPolicy “Always veya On-failure” ise kube-sched container'ı yeniden başlatarak kurtarır ve çalışmasına devam ettirir. **Fakat, sorun node üzerinde çıkarsa, kube-sched, bunu gidip başka bir worker-node'da çalıştırmaz.**

Buna çözüm olarak 3 node tanımlandı, önlerine de bir load balancer konuldu. Eğer birine bir şey olursa diğerleri online olmaya devam edeceği için sorun çözülmüş oldu.
Uygulamanın yeni bir geliştirme sonrasında, tek tek tüm node'lardaki container image'ları yenilenmek zorundadır. Bu noktada karmaşık hale gelen işlerin çözümü "**Deployment**" objesidir.

- Deployment, bir veya birden fazla pod için belirlenen **desired state**'i sürekli **current state**'e getirmeye çalışan bir obje tipidir. Deployment'lar içerisindeki **deployment-controller** ile current state'i desired state'e getirmek için gerekli aksiyonları alır.
- Deployment objesi ile image update etme işlemi tüm nodelarda kolaylıkla yapılabilir.
- Deployment'a işlemler sırasında nasıl davranması gerektiği de (**Rollout**) parametre ile belirtilebilir.
- **Deployment'ta yapılan yeni işlemlerde hata alınırsa, eski haline tek bir komut ile döndürülebilir.**

Deployment oluştururken replica tanımı yapılırsa, k8s cluster'ı her zaman o kadar replika'yı canlı tutmaya çalışacaktır. Manuel olarak deployment’ın oluşturduğu pod’lardan biri silinse bile, arka tarafta yeni bir pod ayağa kaldırılacaktır. Bu nedenle **Singleton Pod** yaratılmamalıdır. Yani, manuel ya da yaml ile direkt pod yaratmayarak bu optimizasyon k8s’e bırakılıyor.

- Tek bir pod yaratılacak olsa bile, bu deployment ile yaratılmalıdır! (k8s resmi önerisi)

Komut ile Deployment Oluşturma

```
kubectl create deployment <deploymentName> --image=<imageName> --replicas=<replicasNumber>
kubectl create deployment <deploymentName> --image=nginx:latest --replicas=2
kubectl get deployment   # Tüm deployment ready kolonuna dikkat!
```

Deployment’taki image’ı Update etme

```
kubectl set image deployment/<deploymentName> <containerName>=<yeniImage>
kubectl set image deployment/firstdeployment nginx=httpd
```

Default strateji olarak, önce bir pod’u yeniler, sonra sırasıyla diğerlerini yeniler. Bu değiştirilebilir.

Deployment Replicas’ını Değiştirme

```
kubectl scale deployment <deploymentName> --replicas=<yeniReplicaSayısı>
```

Deployment Silme

```
kubectl delete deployments <deploymentName>
```

YAML ile Deployment Oluşturma

1. Herhangi bir pod oluşturacak yaml dosyasındaki **metadata** altında kalan satırlar kopyalanır:

```
# podexample.yaml

apiVersion: v1
kind: Pod
metadata:
  name: examplepod
  labels:
    app: frontend
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

2. Deployment oluşturacak yaml dosyasında **template** kısmının altına yapıştırılır. (Indent'lere dikkat!)
3. pod template içerisinden **name** alanı silinir.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: firstdeployment
  labels:
    team: development
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend          # template içerisindeki pod'la eşleşmesi için kullanılacak label.
  template:                  # Oluşturulacak podların özelliklerini belirttiğimiz alan.
    metadata:
      labels:
        app: frontend        # deployment ile eşleşen pod'un label'i.
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80  # dışarı açılacak port.
```

Her deployment'ta **en az bir tane** `selector` tanımı olmalıdır.
Birden fazla deployment yaratılacaksa, farklı label'lar kullanılmalıdır. Deployment'lar kendine ait olan pod'ları karıştırabilir. **Ayrıca, aynı label'ları kullanan singleton bir pod yaratmakta sakıncalıdır!**

ReplicaSet

K8s'de x sayıda pod oluşturan ve bunları yöneten obje türü aslında deployment değildir, tüm bu işleri üstlenen ReplicaSet'tir. Deployment'a istenen derived state söylendiğinde, deployments objesi bir ReplicaSet objesi oluşturur ve tüm bu görevleri ReplicaSet gerçekleştirir.
Bir ReplicaSet'in amacı, herhangi bir zamanda çalışan kararlı bir replika Pod setini sürdürmektir. Genellikle belirli sayıda özdeş Pod'un kullanılabilirliğini garanti eder.

K8s ilk çıktığında **Replication-controller** adında bir obje vardı, halen var ama kullanılmıyor.
`kubectl get replicaset` # Aktif ReplicaSet'leri listeler.
Bir deployment tanımlıyken, üzerinde bir değişiklik yapıldığında; deployment yeni bir ReplicaSet oluşturur ve bu ReplicaSet yeni podları oluşturmaya başlar, bir yandan da eski podlar silinir.

Deployment üzerinde yapılan değişiklikleri geri alma

`kubectl rollout undo deployment <deploymentName>`
Bu durumda ise eski deployment yeniden oluşturulur ve eski ReplicaSet önceki podları oluşturmaya başlar. Bu nedenle, tüm bu işlemleri manuel yönetmemek için doğrudan ReplicaSet oluşturulmaz, işlemler deployment oluşturarak yürütülür.

—> Deployment > ReplicaSet > Pods

ReplicaSet, YAML olarak oluşturulmak istendiğinde tamamen deployment ile aynı şekilde oluşturulur.

Rollout ve Rollback

Rollout ve Rollback kavramları, **deplomen**'ın güncellenmesi sırasında devreye girer, anlam kazanır.
YAML ile deployment tanımı yapılırken 2 tip **`strategy`** seçilir:

Rollout Strategy - `Recreate`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rcdeployment
  labels:
    team: development
spec:
  replicas: 3
  selector:
    matchLabels:
      app: recreate
  strategy:
    type: Recreate  # recreate === Rollout strategy
...
```

"Bu deployment'ta bir değişiklik yapılırsa, öncelikle tüm podları sil, sonrasında yenilerini oluştur."
Bu yöntem daha çok **hardcore migration** yapıldığında kullanılır.
Uygulamanın yeni versionuyla eski versionunun birlikte çalışması **sakıncalı** ise bu yöntem seçilir.
Mesela, bir RabbitMQ consumer'ımız olduğunu varsayalım. Böyle bir uygulamada eski version ve yeni version'un birlikte çalışması genellikle istenmez ve strategy olarak recreate tercih edilir.

Rollback Strategy - `RollingUpdate`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rolldeployment
  labels:
    team: development
spec:
  replicas: 10
  selector:
    matchLabels:
      app: rolling
  strategy:
    type: RollingUpdate         # Rollback Strategy
    rollingUpdate:
      maxUnavailable: 2         # Güncelleme esnasında aynı anda kaç pod silineceği
      maxSurge: 2               # Güncelleme esnasında toplam aktif max pod sayısı
  template:
  ...
```

- Eğer YAML dosyasında strategy belirtilmezse, **default olarak RollingUpdate seçilir**. **maxUnavailable** ve **maxSurge** değerleri ise default **%25**'dir.
- RollingUpdate, Create'in tam tersidir. "Bir değişiklik yapılırsa, hepsini silip; yenilerini oluşturma."

Bu stratejide önemli 2 parametre vardır:

- maxUnavailable –> En fazla silinecek pod sayıdır.

Bir güncellemeye başlandığında en fazla silinecek pod sayısıdır. (%20 de yazabiliriz.)

- maxSurge –> Güncelleme geçişi sırasında sistemde en fazla toplam aktif pod sayısıdır.

Örnek

Image = nginx olan bir deployment olduğunu varsayalım ve komut ile varolan deployment üzerinde nginx image'ı yerine httpd-alphine image'ı güncellemesi yapalım:

`kubectl set image deployment rolldeployment nginx=httpd-alphine --record=true`

`--record=true` parametresi tüm güncelleme aşamalarının kaydedilmesini sağlar. Özellikle, bir önceki duruma geri dönülmek istendiğinde işe yarar.

Yapılan değişikliklerin listelenmesi

`kubectl rollout history deployment rolldeployment` # tüm değişiklik listesini getir.

`kubectl rollout history deployment rolldeployment --revision=2` # spesifik değişikliği gösterir.

Yapılan değişikliklerin geri alınması

`kubectl rollout undo deployment rolldeployment` # bir önceki duruma geri döner.

`kubectl rollout undo deployment rolldeployment --to-revision=1` # Spesifik bir revision'a geri döner.

Canlı olarak deployment güncellemeyi izlemek

`kubectl rollout status deployment rolldeployment -w` # rolldeployment = deploymentName

Deployment'ı güncelleme sırasında pause yapmak

Güncelleme sırasında bir problem çıktı ve geri de dönmek istenmiyorsa, ayrıca sorunun nereden kaynaklandığı da tespit edilmek istendiğinde kullanılır.

`kubectl rollout pause deployment rolldeployment` # rolldeployment = deploymentName

Pause yapılan deployment güncellemesini devam ettirmek

`kubectl rollout resume deployment rolldeployment`

Kubernetes Ağ Yapısı ve Service

Kubernetes Temel Ağ Altyapısı

Aşağıdaki üç kural dahilinde kubernetes network yapısı ele alınmıştır (olmazsa olmazdır):

1- Kubernetes kurulumda pod'lara ip dağıtılması için bir **IP adres aralığı** (--pod-network-cidr) belirlenir.

2- Kubernetes'de her pod bu cidr bloğundan atanacak **uniq bir IP adresine** sahip olur.

3- **Aynı cluster içerisindeki tüm podlar varsayılan olarak birbirleriyle herhangi bir kısıtlama ve NAT olmadan haberleşebilirler.**

Kubernetes içerisindeki container'lar 3 tür haberleşmeye maruz bırakılır:

1- Bir container kubernetes dışındaki bir IP ile haberleşir,

2- Bir container kendisi ile aynı node içerisindeki başka bir container ile haberleşir,

3- Bir container farklı bir node içerisindeki başka bir container ile haberleşir.

İlk 2 senaryoda sorun yok ama 3. senaryo'da NAT konusunda problem yaşanır. Bu sebeple k8s container'ların birbiri ile haberleşme konusunda CNI projesini devreye almıştır.

CNI, yanlızca containerların ağ bağlantısıyla ve containerlar silindiğinde containerlara ayrılan kaynakların drop edilmesiyle ilgilenir.

Kubernetes ise ağ haberleşme konusunda CNI standartlarını kabul etti ve herkesin kendi seçeceği CNI pluginlerinden birini seçmesine izin verdi. CIDR Classless Inter-Domain Routing.

Container NetworkInterface "CNI"

Cloud Native Computing Foundation projesi olan CNI, Linux containerlarda ağ arabirimlerini yapılandırmak için eklentiler yazılabilmesini sağlayan spesifikasyonları belirler. CNI, yalnızca containerların ağ bağlantısıyla ve containerlar silindiğinde ayrılan kaynakların kaldırılmasıyla ilgilenir. Bu odak nedeniyle, CNI geniş bir desteğe sahiptir ve spesifikasyonun uygulanması kolaydır.

Container'ların "Dış Dünya" ile haberleşmesi böyle iken, "Dış Dünya" ise Container'lar ile Service objesiyle haberleşir.

Service

Kubernetes'in network tarafını ele alan objedir.

Service Objesi Çıkış Senaryosu

1 Frontend (React), 1 Backend (Go) oluşan bir sistemimiz olduğunu düşünelim:
Her iki uygulama için birer deployment yazdık ve 3'er pod tanımlanmasını sağladık.
3 Frontend poduna dış dünyadan nasıl erişim sağlayacağım?
Frontend'den gelen istek backend'de işlenmeli, burada çok bir problem yok. Çünkü, her pod'un bir IP adresi var ve k8s içerisindeki her pod birbiriyle bu IP adresleri sayesinde haberleşebilir.
Bu haberleşmenin sağlayabilmesi için; Frontend podları Backend podlarının IP adreslerini bilmelidir. Bunu çözmek için, frontend deployment'ına tüm backend podlarının IP adreslerini yazdım. Fakat, pod'lar güncellenebilir, değişebilir ve bu güncellemeler esnasında yeni bir IP adresi alabilir. Yeni oluşan IP adreslerini tekrar Frontend deployment'ında tanımlamam gerekir.

İşte tüm bunları çözmek için Service objesi yaratılır. K8s, Pod'lara kendi IP adreslerini ve bir dizi Pod için tek bir DNS adı verir ve bunlar arasında yük dengeler.

Service Tipleri

ClusterIP (Container'lar Arası)

- Bir ClusterIP service'i yaratıp, label'lar sayesinde podlarla ilişkilendirebilir. Bu obje yaratıldığında, Cluster içerisindeki tüm podların çözebileceği unique bir DNS adresi olur. Ayrıca, her k8s kurulumunda sanal bir IP range'e sahip olur. (Ör: 10.0.100.0/16)
- ClusterIP service objesi yaratıldığı zaman, bu objeye bu IP aralığından **bir IP atanır** ve **ismiyle bu IP adresi Cluster'ın DNS mekanizmasına kaydedilir.** Bu IP adresi **Virtual (Sanal) bir IP adresidir.**
- Kube-proxy tarafından tüm node'lardaki IP Table'a bu IP adresi eklenir. Bu IP adresine gelen her trafik Round Troppin algoritmasıyla Pod'lara yönlendirilir. Böylece; Frontend node'larına bu IP adresi tanımlanarak (ismine de backend denirse), backend'e erişim için bu IP adresinin kullanılması sağlanabilir. (**Selector = app:backend**) Tek tek her defasında Frontend podlarına Backend podlarının IP adresleri eklenmek zorunda kalınmaz.

ClusterIP; Container'lar arasında Service Discovery ve Load Balancing görevi üstlenir!

NodePort Service (Dış Dünya -> Container)

- Bu service tipi, Cluster dışından gelen bağlantıları çözmek için kullanılır.
- NodePort key'i kullanılır.

LoadBalancer Service (Cloud Service'leri İçin)

- Bu service tipi, sadece Agent K8s, Google K8s Engine gibi yerlerde kullanılır.

ExternalName Service (Şimdilik gereksizdir.)

Service Objesi Uygulaması

Cluster IP Örneği

```
apiVersion: v1
kind: Service
metadata:
  name: backend # Service ismi
spec:
  type: ClusterIP # Service tipi
  selector:
    app: backend # Hangi podla ile eşleşeceği (pod'daki label ile aynı)
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
```

Herhangi bir obje, cluster içerisinde oluşan **`clusterIP:5000`** e istek attığında karşılık bulur.

Önemli Not: Service'lerin ismi oluşturulduğunda şu formatta olur:
`serviceName.namespaceName.svc.cluster.domain`

Eğer aynı namespace'de başka bir object bu servise gitmek istese core DNS çözümlemesi sayesinde direkt **`backend`** yazabilir. Başka bir namespaceden herhangi bir object ise ancak yukarıdaki **uzun ismi** kullanarak bu servise ulaşmalıdır.

NodePort Örneği

- Unutulmamalıdır ki, tüm oluşan NodePort servislerinin de **ClusterIP**'si mevcuttur. Yani, içeriden bu ClusterIP kullanılarak bu servisle konuşulur.
- **`minikube service –url <serviceName>`** ile minikube kullanırken tunnel açabiliriz. Çünkü, normalde bu worker node'un içerisine dışarıdan erişilemez. Bu tamamen `minikube` ile alakalıdır.

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Load Balancer Örneği

Google Cloud Service, Azure üzerinde oluşturulan cluster'larda çalışacak bir servistir.

```
apiVersion: v1
kind: Service
metadata:
  name: frontendlb
spec:
  type: LoadBalancer
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Imperative Olarak Service Oluşturma

```
kubectl delete service <serviceName>        # Service siler
kubectl expose deployment backend --type=ClusterIP --name=backend     # ClusterIP Service yaratır
kubectl expose deployment backend --type=NodePort --name=backend      # NodePort Service yaratır
```

Endpoint Nedir?

Nasıl deployment'lar aslında ReplicaSet oluşturdu ise, Service objeleri de arka planda birer Endpoint oluşturur. Service'lere gelen isteklerin nereye yönleneceği Endpoint'ler ile belirlenir.

`kubectl get endpoints`

Bir pod silindiğinde yeni oluşacak pod için, yeni bir endpoint oluşturulur.

Liveness Probes

Bazen container içerisinde çalışan bir uygulama, düzgün çalışmayabilir. Uygulama çökmemiş, kapanmamış ama aynı zamanda işlevini tam yerine getirmiyorsa kubelet bunu tespit edemiyor. Bir uygulama çalışıyor görünmesine rağmen, zombi olabilir. Bu durumda bir container'ı yeniden başlatmak, hatalara rağmen uygulamayı kullanılabilir hale getirebilir. Liveness; container'a bir request göndererek, TCP connection açarak veya container içerisinde bir komut çalıştırarak doğru çalışıp çalışmadığını anlamayı sağlar.
Kubelet, bir container'ın ne zaman yeniden başlatılacağını belirlemek için liveness probe'ları kullanır.

```
# http get request gönderildiğinde; 200 ve üzeri cevap dönerse başarılı! dönmezse kubelet container'ı yeniden başlatacak.
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:                # get request'i gönderiyoruz.
        path: /healthz        # path tanımı
        port: 8080            # port tanımı
        httpHeaders:          # get request'imize header eklemek istersek
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3  # uygulama hemen ayağa kalkmayabilir,çalıştıktan x sn sonra isteği gönder.
      periodSeconds: 3        # kaç sn'de bir bu istek gönderilecek. (healthcheck test sürekli yapılır.)
---
# uygulama içerisinde komut çalıştırıldığında; eğer exit -1 sonucu alınırsa container yeniden başlatılır.
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox



    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:                     # komut çalıştırılır.
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
---
# tcp connection yaratılır. Eğer başarılıysa devam eder, yoksa container baştan başlatılır.
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    livenessProbe:          # tcp connection yaratılır.
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

Readiness Probes

Deployment Güncelleme Senaryosu ve Readiness Probe

Deployment güncellendi "v2 imaji".
Yeni imajla yeni bir pod (v2) oluşturuldu.
Yeni pod (v2) çalışmaya başladı.
readiness check mekanizması çalışmaya başladı. initialDelaySeconds kadar bekledi ardından ilk kontrolünü yaptı.
Kontrol sonucu olumlu olduğu anda pod (v2) service'e eklendi.
Bu noktada eski pod'un (v1) service ile bağlantısı kesildi.
Eski pod (v1) "eğer işlemesi gereken istekler varsa işlemeye devam etmesi için" henüz kapatılmadı.
Eski pod'un (v1) kendini düzgün kapatması için sistem sigterm sinyali gönderdi.
Pod (v1) üzerindeki işlemleri tamamladıktan sonra kendisi kapattı.

Uygulama çalışıyor ama hizmet sunmaya hazır olmayabilir.

Kubelet, bir container'ın trafiği kabul etmeye (Initial status) hazır olduğunu belirlemek için **Readiness Probes** kullanır. Bir Pod'daki tüm container'lar Readiness Probes kontrolünden onay alırsa **Service Pod'un arkasına eklenir.**

Yeni image'lar oluşturulurken eski Pod'lar hemen terminate edilmez. Çünkü, içerisinde daha önceden alınmış istekler ve bu istekleri işlemek için yürütülen işlemler olabilir, k8s önce bu Pod'un service ile ilişkisini keser ve yeni istekler almasını engeller, mevcut isteklerin de sonlanmasını bekler.

`terminationGracePeriodSconds: 30` Mevcut işlemler biter, 30 sn bekler ve kapanır.

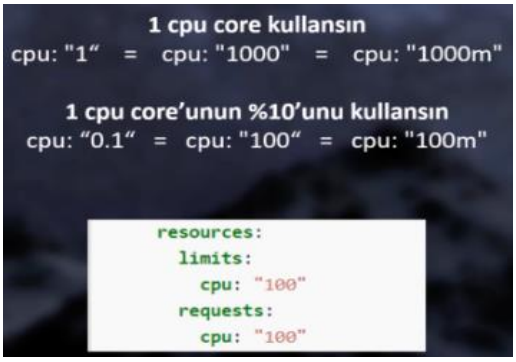
Readiness ilk çalışma anını baz alırken, Liveness sürekli çalışıp çalışmadığını kontrol eder.
Mesela; Backend'in ilk açılışta MongoDB'ye bağlanması için geçen bir süre vardır. MongoDB bağlantısı sağlandıktan sonra Pod'un arkasına Service eklenmesi mantıklıdır. Bu sebeple, burada readiness kullanılabilir.
Liveness gibi 3 farklı yöntem var: **http/get**, **tcp connection** ve **command çalıştırma.**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    team: development
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: ozgurozturknet/k8s:blue
        ports:
        - containerPort: 80
        livenessProbe:
          httpGet:
            path: /healthcheck
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
        readinessProbe:
          httpGet:
            path: /ready                          # Bu endpoint'e istek atılır, OK dönerse uygulama çalıştırılır.
            port: 80
          initialDelaySeconds: 20                 # Başlangıçtan 20 sn gecikmeden sonra ilk kontrol yapılır.
          periodSeconds: 3                        # 3sn'de bir denemeye devam eder.
          terminationGracePeriodSconds: 50        # Sonlandırma bekleme süresi
---
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

`terminationGracePeriodSeconds`: Sonlandırma bekleme süresi, bir pod terminet edilmek istendiğinde, pid 1'e SIGKILL (yaptığın ne varsa bırak ve kapan) gönderilmez, bunun yerine SIGTERM (eğer yaptığın işler varsa onları tamamla ve bitince de kapan) gönderilir. Bu süre sonunda kapanmazsa SIGKILL gönderilerek zorla kapatılır.

Resource Limits

Pod'ların CPU ve Memory ayarlamalarını yönetmemiyi sağlar. Aksi belirtilmedikçe k8s üzerinde çalıştığı makinenin CPU ve Memory'sini %100 kullanabilir. Bu durum bir sorun oluşturabileceği için Pod'ların CPU ve Memory kullanamı sınırlandırılır.



CPU

CPU kaynağı, CPU birimleri olarak ölçülür.
Kubernetes'te 1 CPU şuna eşdeğerdir:
1 AWS VCPU
1 GCP Core
1 Azure vCore
1 Hyperthread bare-metal
Kesirli değerlere izin verilir.
0,5 CPU talep eden bir container,
1 CPU talep eden bir containerin yarısı kadar CPU alır.
m son eki mili anlamında; 100m CPU, 100 milliCPU ve 0.1 CPU aynıdır. 1m'den daha ince hassasiyete izin verilmez. CPU her zaman mutlak bir miktar olarak talep edilir, asla göreli bir miktar olarak talep edilmez; 0.1; tek çekirdekli, çift çekirdekli veya 48 çekirdekli bir makinede aynı miktarda CPU'dur.

MEMORY

Cluster'da mevcut durumda kullanılabilir bellek varsa,
bir container belirlenen bellek isteğini aşabilir.
Ancak bir container bellek sınırından fazlasını kullanamaz.
Bir container, sınırından daha fazla bellek ayırırsa,
container sonlandırma için aday olur.
Container,
limitinin ötesinde bellek tüketmeye devam ederse, sonlandırılır.
Sonlandırılmış bir container yeniden başlatılabiliyorsa,
diğer herhangi bir runtime hatası türünde olduğu gibi
kubelet onu yeniden başlatır.

YAML Dosyasında Tanım

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: requestlimit
  name: requestlimit
spec:
  containers:
  - name: requestlimit
    image: ozgurozturknet/stress
    resources:
      requests:                 # Podun çalışması için en az gereken gereksinim
        memory: "64M"           # Bu podu en az 64M 250m (yani çeyrek core)
        cpu: "250m"             # = Çeyrek CPU core = "0.25"
      limits:                   # Podun çalışması için en fazla gereken limit
        memory: "256M"
        cpu: "0.5"              # = "Yarım CPU Core" = "500m"
```

- Eğer gereksinimler sağlanamazsa container oluşturulamaz.
- Memory, CPU'ya göre farklı çalışır. Memory limitden fazla değer istediğinde engellenmez. Eğer pod, limitden fazla memory'e ihtiyaç duyarsa "**OOMKilled**" durumuna geçerek restart edilir.

Environment Variables

Bir node.js sunucusu oluşturulduğunu ve veritabanı bilgilerinin sunucu dosyaları içerisinde saklandığını düşünelim. Sunucu dosyalarından oluşturulan container image'ı büyük bir güvenlik açığı oluşturur. Bu nedenle Environment Variables kullanılır.

YAML Tanımlaması

```
apiVersion: v1
kind: Pod
metadata:
  name: envpod
  labels:
    app: frontend
spec:
  containers:
  - name: envpod
    image: ozgurozturknet/env:latest
    ports:
    - containerPort: 80
    env:
      - name: USER          # önce name'ini giriyoruz.
        value: "Ozgur"      # sonra value'sunu giriyoruz.
      - name: database
        value: "testdb.example.com"
```

`kubectl exec <podName> -- printenv` # Pod içinde tanımlanmış ortam değişkenlerini gösterir

Port-Forward (Local -> Pod)

Local sunuculardan k8s cluster'ı içerisindeki objelere direkt ulaşabilmek için **port-forward** yapılabilir. Bu objeyi test etmek için en iyi yöntemlerden biridir.

`kubectl port-forward <objectType>/<podName> <localMachinePort>:<podPort>`

`kubectl port-forward pod/envpod 8080:80` # Local 8080 portuna gelen tüm istekleri,bu podun 80 portuna gönder.

`curl 127.0.0.1:8080` # Test için yazılabilir.

`Ctrl + C` yapıldığında port-forwarding sona erer.

Volume

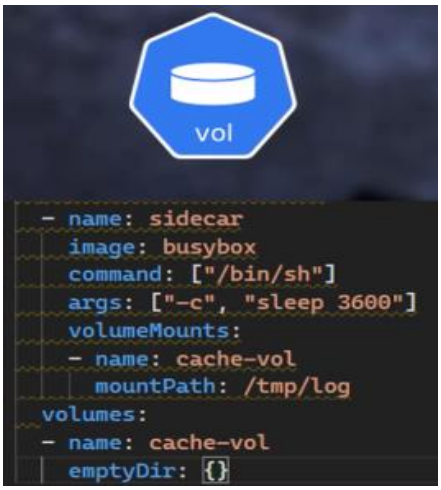
- Container'ın dosyaları container varolduğu sürece bir dosya içerisinde tutulur. Eğer container silinirse, bu dosyalarda birlikte silinir. Her yeni bir container yaratıldığında, container'a ait dosyalar yeniden yaratılır. (Stateless kavramı)
- Bazı containerlarda bu dosyaların silinmemesi (tutulması) gerekebilir. İşte bu durumda Ephemeral (Geçici) Volume kavramı devreye giriyor. DB container'ında data kayıplarına uğramamak için volume kurgusu yapılmalıdır. (Stateful kavramı)
- Ephemeral Volume'ler aynı pod içerisindeki tüm containerlara aynı anda bağlanabilir.
- Diğer bir amaç ise aynı pod içerisindeki birden fazla container'ın ortak kullanabileceği bir alan yaratmaktır.

Ephemeral Volume'lerde Pod silinirse tüm veriler kaybolur. Fakat, container silinip tekrar yaratılırsa Pod'a bir şey olmadığı sürece veriler saklanır. İki tip Ephemeral Volume vardır:

emptyDir Volume

emptyDir volume ilk olarak bir Pod bir node'a atandığında oluşturulur ve bu Pod o node'da çalıştığı sürece var olur. Adından da anlaşılacağı gibi, emptyDir birimi başlangıçta boştur.
Pod içindeki tüm containerlar, emptyDir volume'deki aynı dosyaları okuyabilir ve yazabilir, ancak bu birim her kapsayıcıda aynı veya farklı yollara mount edilebilir.
Bir Pod herhangi bir nedenle bir node'dan silindiğinde, emptyDir içindeki veriler de kalıcı olarak silinir.

```
apiVersion: v1
kind: Pod
metadata:
  name: emptydir
spec:
  containers:
  - name: frontend
    image: ozgurozturknet/k8s:blue
    ports:
    - containerPort: 80
    livenessProbe:
      httpGet:
        path: /healthcheck
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
    volumeMounts:
    - name: cache-vol                  # volume ile bağlantı sağlamak için.
      mountPath: /cache                # volume içerisindeki hangi dosya?
  - name: sidecar
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "sleep 3600"]
    volumeMounts:
    - name: cache-vol
      mountPath: /tmp/log              # volume içerisindeki hangi dosya?
  volumes:
  - name: cache-vol                    # Önce volume'ü oluşturur sonra container'lara tanımlarız.
    emptyDir: {}
```



hostPath Volume

Bir hostPath volume, worker node dosya sisteminden Pod'a bir dosya veya dizini bağlayabilme imkanı verir. Çoğu Pod ihtiyaç duymaz, ancak bazı uygulamalar için güçlü bir kaçış kapısıdır.

- Çok nadir durumlarda kullanılır, kullanılırken dikkat edilmelidir.
- emptyDir'da volume klasörü pod içerisinde yaratılırken, hostPath'te volume klasörü worker node içerisinde yaratılır.
- Üç farklı tip'te kullanılır:
  - **Directory :** Worker node üzerinde zaten var olan klasörler için kullanılır.
  - **DirectoryOrCreate :** Zaten var olan klasörler ya da bu klasör yoksa yaratılması için kullanılır.
  - **FileOrCreate :** Klasör değil! Tek bir dosya için kullanılır. Yoksa yaratılır.

```
apiVersion: v1
kind: Pod
metadata:
  name: hostpath
spec:
  containers:
  - name: hostpathcontainer
    image: ozgurozturknet/k8s:blue
    ports:
    - containerPort: 80
    livenessProbe:
      httpGet:
        path: /healthcheck
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
    volumeMounts:
    - name: directory-vol
      mountPath: /dir1
    - name: dircreate-vol
      mountPath: /cache
    - name: file-vol
      mountPath: /cache/config.json
  volumes:
  - name: directory-vol
    hostPath:
      path: /tmp
      type: Directory
  - name: dircreate-vol
    hostPath:
      path: /cache
      type: DirectoryOrCreate
  - name: file-vol
    hostPath:
      path: /cache/config.json
      type: FileOrCreate
```

Secret

- Hassas bilgileri (user, password gibi) saklamak için Environment Variables kullanılabilse de, bu yöntem ideal olmayabilir.
- Secret objesi hassas bilgileri uygulamanın obje tanımlarının yapıldığı YAML dosyalarından ayırır ve ayrıca yönetilmesini sağlar.
- Token, username, password gibi bilgileri secret ile saklamak her zaman daha güvenli ve esnektir.

Declarative Secret Oluşturma

- Atanacak secret ile oluşturulacak pod'lar aynı namespace üzerinde olmalıdır.
- 8 farklı tipte secret oluşturulabilir. Opaque generic bir type'dır ve hemen hemen her hassas data bu tipi kullanarak saklanabilir.

Örnek bir secret.yaml dosyası:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:                    # Hassas data'ları stringData altına yazıyoruz.
  db_server: db.example.com
  db_username: admin
  db_password: P@ssw0rd!
```

```
kubectl describe secrets <secretName>      # Secret içerisindeki dataları gösterir.
```

Imperative Secret Oluşturma

```
kubectl create secret generic <secretName> --from-literal=db_server=db.example.com --from-literal=db_username=admin
```

Bu komutdaki **generic** parametresi, yaml dosyasındaki **Opaque** parametresine denktir.

Eğer CLI'de hassas verilerin girilmesi istenmiyorsa, her bir veri ayrı bir .txt dosyasına girilerek; **`–from-file=db_server=server.txt`** komutu çalıştırılabilir.
Ayrıca .txt yerine .json'da kullanılabilir. O zaman **`–from-file=config.json`** yazılmalıdır.
Json örneği:

```
{
    "apiKey": "6bba108d4b2212f2c30c71dfa279e1f77cc5c3b2",
}
```

Pod'dan Secret'ı Okuma

Oluşturulan Secret'ı Pod'a aktarmak için, Volume olarak aktarma ve Environment Variable olarak aktarma olmak üzere iki yöntem vardır. Her iki yöntem de aşağıdaki YAML dosyasında gösterilmektedir:

```
apiVersion: v1
kind: Pod
metadata:
  name: secretpodvolume
spec:
  containers:
  - name: secretcontainer
    image: ozgurozturknet/k8s:blue
    volumeMounts:                    # 2) Oluşturulan secret'lı volume'ü pod'a dahil ediyoruz.
    - name: secret-vol
      mountPath: /secret             # 3) Uygulama içerisinde /secret klasörü volume'e dahil olmuştur.
                                     # Artık uygulama içerisinden bu dosyaya ulaşıp, değerleri okuyabiliriz.
  volumes:                           # 1) Önce volume'ü oluşturuyoruz ve secret'ı volume'e dahil ediyoruz.
  - name: secret-vol
    secret:
      secretName: mysecret3
---
                                     # Bu pod'a exec ile girildiğinde root altındaki secret klasöründeki dosyaların ismi "KEY", içindeki değerler "VALUE"dur.
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secretpodenv
spec:
  containers:
  - name: secretcontainer
    image: ozgurozturknet/k8s:blue
    env:                              # Tüm secretları pod içerisinde env. variable olarak tanımlayabiliriz, tüm secretları ve değerlerini tek tek tanımladık.
      - name: username
        valueFrom:
          secretKeyRef:
            name: mysecret3           # mysecret3 isimli secret'ın "db_username" key'li değerini al.
            key: db_username
      - name: password
        valueFrom:
          secretKeyRef:
            name: mysecret3
            key: db_password
      - name: server
        valueFrom:
          secretKeyRef:
            name: mysecret3
            key: db_server
---
apiVersion: v1
kind: Pod
metadata:
  name: secretpodenvall
spec:
  containers:
  - name: secretcontainer
    image: ozgurozturknet/k8s:blue
    envFrom:                          # 2. yöntemle aynı, sadece tek fark tüm secretları tek bir seferde tanımlıyoruz.
    - secretRef:
        name: mysecret3
```

ConfigMap

ConfigMap, gizli olmayan verileri anahtar/değer eşlenikleri olarak depolamak için kullanılan bir API nesnesidir. Podlar, ConfigMap'i environment variable, komut satırı argümanları veya bir volume olarak bağlanan yapılandırma dosyaları olarak kullanabilir.

ConfigMap'ler Secret objeleriyle tamamen aynı mantıkta çalışır. Tek farkı; Secret'lar etcd üzerinde base64 ile encoded edilerek encrypted bir şekilde saklanır, ConfigMap'ler ise encrypted edilmez ve bu sebeple hassas datalar içermemelidir.

- Pod içerisine Volume veya Environment Variables olarak tanımlayabiliriz.
- Oluşturulma yöntemleri Secret ile aynı olduğundan aynı komutlar burada da geçerlidir.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:                 # Key-Value şeklinde girilmelidir.
  db_server: "db.example.com"
  database: "mydatabase"
  site.settings: |    # Birden fazla satır yazımı için "|" kullanılır.
    color=blue
    padding:25px
---
apiVersion: v1
kind: Pod
metadata:
  name: configmappod
spec:
  containers:
  - name: configmapcontainer
    image: ozgurozturknet/k8s:blue
    env:
      - name: DB_SERVER
        valueFrom:
          configMapKeyRef:
            name: myconfigmap
            key: db_server
      - name: DATABASE
        valueFrom:
          configMapKeyRef:
            name: myconfigmap
            key: database
    volumeMounts:
      - name: config-vol
        mountPath: "/config"
        readOnly: true
  volumes:
    - name: config-vol
      configMap:
        name: myconfigmap
```

config.yaml Dosyasından ConfigMap Oluşturma

Uygulama içerisinde kullanılmak üzere her bir ortam için (QA, SIT ve PROD) config.qa.yaml veya config.prod.json dosyası olsun. CI/CD içerisinde bu ortamlara göre çalışacak doğru config dosyasından ConfigMap oluşturma:

```
// config.json
{
  "name": "Pamuk",
  "surName": "Koza",
  "email": "pamuk@banka.com",
  "apiKey": "6bba108d4b2212f2c30c71dfa279e1f77cc5c3b2",
  "text": ["ali", "veli", "on", "yedi", 9, false]
}
```

1. Kubectl ile config.json dosyasından ConfigMap objesi oluşturalım.
Eğer bir CI/CD üzerinde çalıştırılıyor ve logları takip ediliyorsa;

--dry-run normalde deprecated, fakat hala eski versionlarda kullanılıyor.

```
kubectl create configmap xyzconfig --from-file ${configFile} -o yaml --dry-run | kubectl apply -f -
```

yeni hali --dry-run="client"

```
kubectl create configmap xyzconfig --from-file config.json -o yaml --dry-run="client" | kubectl apply -f -
```

Sondaki "-" (dash) çıkan pipe'ın ilk tarafından çıkan output'u alır.

- Komut çalıştırıldığında kubectl config.json dosyasını alır; `kubectl apply` komutunun kullanabileceği bir ConfigMap YAML dosyası içeriği oluşturur ve bu içeriği **–dry-run** seçeneğinden dolayı output olarak ekrana basar.
- Output | ile `kubectl apply` komutuna eklenmesi ile cluster'a gönderiliyor ve ConfigMap tanıtımı yapılıyor.

Eğer logları okumadan direkt config.json dosyasından bir ConfigMap yaratmak isteniyorsa;

config.json yerine bir çok formatta dosya kullanılabilir: ÖR: yaml

```
kubectl create configmap <configName> --from-file config.json
```

2. Pod'u oluştururken ConfigMap'deki değerleri Pod içerisinde yer alan bir klasördeki bir dosyaya aktarmak ve bunu "volume" mantığında bir dosya içerisinde saklamak gerekiyor.
- “volumes” kısmında volume ve configMap tanımlanır.
- Pod içerisindeki “volumeMounts” kısmında ise tanımlanan bu volume pod'a bağlanır.
  - **mountPath**, configMap içerisinde yer alan dosyanın, Pod içerisinde hangi klasör altına kopyalanacağı ve yeni isminin ne olacağı belirtilir.
  - **subPath** ile configMap içerisindeki dosyanın (ÖR: `config.json`) ismini vermek gerekiyor. Böylece pod yaratılırken “Bu dosyanın ismi değişecek.” deniliyor.

```
apiVersion: v1
kind: Pod
metadata:
  name: configmappod4
spec:
  containers:
  - name: configmapcontainer
    image: ozgurozturknet/k8s:blue
    volumeMounts:
      - name: config-vol
        mountPath: "/config/newconfig.json"
        subPath: "config.json"
        readOnly: true
  volumes:
    - name: config-vol
      configMap:
        name: test-config  # Bu ConfigMap içerisinde config.json dosyası vardır.
```

volumes segmentinin alternatif biçimi

```
  volumes:
    - name: config-vol
      projected:
        sources:
        - configMap:
            name: test-config
            items:
              - key: config.json
                path: config.json
```

Ingress

Uygulamanın dış dünyaya erişebilmesi ve/veya dış dünyanın da uygulamya erişilebilmesi için kullanılan yapıdır.

Örnek Senaryo - 1
Azure gibi bir cloud service kullanıldığını ve servis içerisinde bir LoadBalancer service tanımlandığını varsayalım. Azure, bu LoadBalancer servisine bir IP atar ve bu IP'ye gelen tüm istekleri bu LoadBalancer karşılar. DNS'te bu IP adresi ile domain eşleştirilerek, istemcilerin kolay bir şekilde erişmesi sağlanır.
Aynı k8s cluster içerisinde bir tane daha app ve aynı servislerin tanımladığını düşünelim. Hatta abartılı da olsa 2, 3, 4 derken her bir LoadBalancer için **Azure'a ayrıca ödeme ve ayarlarının manuel yapılması gerekiyor.**

Örnek Senaryo - 2 (Path-based Routing)
Bu kurguda ise; kullanıcı **example.com**'a girdiğinde A uygulaması, **example.com/contact**'a girdiğinde ise B uygulaması çalışacaksa, DNS'te **/contact** path'i tanımlanamayacağı için LoadBalancer ile kurgulanamaz. Burada bir gateway gibi çalışan; kullanıcıyı her koşulda karşılayan bir load balancer ihtiyacı vardır.
İşte bütün bunlar **Ingress Controller** ve **Ingress Object** ile çözülüyor:

Ingress controller

Ingress controller, L7 Application Loadbalancer kavramının Kubernetes spesifikasyonlarına göre çalışan ve Kubernetes'e deploy ederek kullanabilen türüdür.
Nginx, HAproxy, Traefik en bilinen ingress controller uygulamalarıdır.

- Ingress Controller, Nginx, Traefik, KrakenD gibi kullanılabilecek bir load balancer uygulamasıdır. Bu uygulamalardan biri seçilerek; k8s cluster'a deploy edilebilir ve LoadBalancer servisi kurularak dışarıya expose edilebilir. Böylece, uygulama public bir IP'ye sahip olur ve dış dünya ile tamamen bu IP üzerinden iletişim kurulabilir.
- Gelen istekleri yönlendirme sırasında **Ingress Objeler** devreye girer. Ingress Controller'larda yapılacak konfigürasyonlarla Ingress Object'leri ve Ingress Controller'ların gelen isteklere karşı nasıl davranması gerektiği belirlenir.
- Load balancing, **SSL termination** ve **path-name based routing** gibi özelliklere sahiptir.

Local Ingress Uygulaması

1) minikube Ayarları

- Ingress çalıştırmak için minikube driver'ı değiştirmek gerekir;
  - Windows için **Hyper-V**, macOS ve Linux için **VirtualBox** seçilebilir. Kurulu olması gerekir.

```
minikube start --driver=hyperv
```

2) Ingress Controller Seçimi ve Kurulumu

- Her ingress controller kurulumu farklıdır ve nginx ile devam edilecektir. Kurulum detayları her uygulamanın kendi web sitesinden öğrenilebilir.
  Kurulum detayları; https://kubernetes.github.io/ingress-nginx/deploy/
- minikube, yoğun olarak kullanılan nginx gibi bazı ingress controller'ları daha hızlı aktif edebilmek için addon olarak sunmaktadır.

```
minikube addons enable ingress  # ingress addonunu aktif eder.
minikube addons list  # tüm addon'ları listeler.
```

- **Nginx** kurulduğu zaman kendisine **ingress-nginx** adında bir **namespace** yaratır.

```
kubectl get all -n ingress-nginx   # ingress-nginx namespace'ine ait tüm objeleri listeler
```

3) Ingress Uygulamasını Deploy Etme

- **blueapp**, **greenapp**, **todoapp** için hem pod'ları hem de service'leri yaratan yaml dosyası deploy edilir.
- Tüm service'lerin ClusterIP tipinde birer service olduğu unutulmamalıdır.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: blueapp
  labels:
    app: blue
spec:
  replicas: 2
  selector:
    matchLabels:
      app: blue
  template:
    metadata:
      labels:
        app: blue
    spec:
      containers:
      - name: blueapp
        image: ozgurozturknet/k8s:blue
        ports:
        - containerPort: 80
        livenessProbe:
          httpGet:
            path: /healthcheck
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
        readinessProbe:
          httpGet:
            path: /ready
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 3
---


apiVersion: v1
kind: Service
metadata:
  name: bluesvc
spec:
  selector:
    app: blue
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: greenapp
  labels:
    app: green
spec:
  replicas: 2
  selector:
    matchLabels:
      app: green
  template:
    metadata:
      labels:
        app: green
    spec:
      containers:
      - name: greenapp
        image: ozgurozturknet/k8s:green
        ports:
        - containerPort: 80
        livenessProbe:
          httpGet:
            path: /healthcheck
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
        readinessProbe:
          httpGet:
            path: /ready
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 3
---


apiVersion: v1
kind: Service
metadata:
  name: greensvc
spec:
  selector:
    app: green
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: todoapp
  labels:
    app: todo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: todo
  template:
    metadata:
      labels:
        app: todo
    spec:
      containers:
      - name: todoapp
        image: ozgurozturknet/samplewebapp:latest
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: todosvc
spec:
  selector:
    app: todo
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

4) Ingress Objelerini Deploy Etme ve Ayarlama

- Load balancer için gerekli olan Ingress Controller Nginx olarak seçildi ve kuruldu.
- Her bir app için gerekli olan ClusterIP tipinde servisleri de kurduktan sonra, sıra istemcilerin **example.com/a** ile **A** service'ine gitmesi için gerekli **Ingress obje**lerini de deploy etmeye geldi.

blue, green app'ler için Ingress Obje tanımlama:

- `pathType` kısmı `exact` veya `Prefix` olarak 2 şekilde ayarlanabilir.

Detaylı bilgi için: https://kubernetes.io/docs/concepts/services-networking/ingress/

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: appingress
  annotations:
  # Nginx üzerinde ayarlar, annotations üzerinden yapılır.
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
    - host: k8sfundamentals.com
      http:
        paths:
          - path: /blue
            pathType: Prefix
            backend:
              service:
                name: bluesvc
                port:
                  number: 80
          - path: /green
            pathType: Prefix
            backend:
              service:
                name: greensvc
                port:
                  number: 80
```

- Farklı bir `path` kullanarak hazırlanan Ingress Objesi:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: todoingress
spec:
  rules:
    - host: todoapp.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: todosvc
                port:
                  number: 80
```

5) Tanımlanan Ingress Objelerini Test:

`kubectl get ingress`

- URL'ler ile simülasyon için, hosts dosyasını düzenlemek gerekir.