Kubernetes Temelleri

Container Orchestration Nedir?

Konteynerleştirilmiş uygulamaların dağıtımını, yönetimini ve ölçeklenmesini otomatikleştiren bir teknolojidir.

Kubernetes Nedir?

Kubernetes, hem beyan temelli yapılandırmayı hem de otomasyonu kolaylaştıran, container yüklerini ve hizmetleri yönetmek için oluşturulmuş, taşınabilir ve genişletilebilir açık kaynaklı bir platformdur.

Bilişim sektöründe en yaygın kullanılan **container orchestrator** olup, uygulamaların yönetimini kolaylaştırır. Kubernetes sayesinde, **"önce şunu yap, sonra bunu yap"** yerine, **"şöyle bir yapı istiyorum"** şeklinde bir tanım yapılır. Kubernetes, neyin olması gerektiğini bilir ve kendisi en uygun şekilde uygular.

Kubernetes Komponentleri

Monolith vs Microservice

- **Monolith:** Tüm uygulamanın tek bir yapı içinde çalıştığı geleneksel yazılım mimarisi.
- **Microservice:** Uygulamanın küçük, bağımsız servisler halinde çalıştığı ve her servisin belirli bir işlevi yerine getirdiği mimari.

Kubernetes de bir **microservice mimarisi** gibi çalışır. Tek bir "Kubernetes" uygulaması yoktur, aksine Kubernetes platformu, birbirleriyle entegre çalışan birçok servisten oluşur.

Control Plane (Kontrol Düzlemi)

Control Plane, Kubernetes cluster’ını yöneten ve yönlendiren merkezi bileşenler kümesidir.

- **kube-apiserver:** Kubernetes API’sini ortaya çıkaran ve cluster’ın yönetim giriş noktası olan en kritik bileşendir. Tüm diğer bileşenler ve worker node'lar, bu API ile iletişim kurar.
- **etcd:** Tüm cluster verisini, metadata bilgilerini ve Kubernetes’de oluşturulan objeleri saklayan, **dağıtılmış anahtar-değer veri deposu**.
- **kube-scheduler:** Yeni oluşturulan ya da bir node ataması yapılmamış Pod’ları izler ve uygun bir **worker node** belirleyerek bu podları o node'a yerleştirir.
- **kube-controller-manager:** Kubernetes içinde çalışan çeşitli controller’ları yöneten bileşendir. Mantıksal olarak her controller ayrı bir süreçtir, ancak hepsi tek bir binary olarak çalıştırılır. Önemli controller’lar:
  - **Node Controller:** Node’ların durumunu izler ve arızalanan node’ları tespit eder.
  - **Job Controller:** Tek seferlik ve belirli sayıda çalışması gereken işler için pod yönetimini sağlar.
  - **Service Account & Token Controller:** Kubernetes içindeki servis hesaplarını ve kimlik doğrulama token’larını yönetir.
  - **Endpoints Controller:** Servislerin bağlı olduğu pod’ları güncelleyerek, trafiğin doğru yönlendirilmesini sağlar.

Worker Nodes (İşçi Düğümleri)

Worker node'lar, uygulama pod’larının çalıştığı makineler veya sanal sunuculardır. Temel bileşenleri şunlardır:

- **kubelet:** Node içinde çalışan ve Kubernetes API ile iletişim kuran ana bileşen. Pod’ların yaşam döngüsünü yönetir.
- **kube-proxy:** Node içinde ağ trafiğini yöneten bileşen. Kubernetes servisleri arasındaki iletişimi sağlar.
- **Container Runtime:** Pod’ları çalıştıran container motorudur (örn: Docker, containerd, CRI-O).

Kurulum

- **Chocolatey:** Windows için bir paket yöneticisidir. Yazılım ve araçları komut satırından kolayca yüklemeye, güncellemeye ve kaldırmaya yarar.
- **kubectl:** Kubernetes cluster’larını yönetmek için kullanılan komut satırı aracıdır. Pod oluşturma, servis yönetme, logları görüntüleme gibi işlemleri yapar.
- **Minikube:** Lokal bilgisayarda tek node'lu bir Kubernetes cluster'ı çalıştırmaya yarayan hafif bir araçtır. Kubernetes'i öğrenmek, test etmek ve geliştirme yapmak için kullanılır.

- **Minikube Kurulumu (Chocolatey ile)**

```
choco install minikube -y
```

- **Minikube Başlatma**→ Minikube, varsayılan olarak bir sanal makine içinde tek node’lu bir Kubernetes cluster’ı başlatır.

```
minikube start
```

- → Minikube, varsayılan olarak bir sanal makine içinde tek node’lu bir Kubernetes cluster’ı başlatır.

Kubernetes Komutları ve Açıklamaları

Kubeconfig Yönetimi

Kubernetes'te
context, küme bağlantı ayarlarını içeren ve belirli bir **kullanıcı**, **küme (cluster)** ve **ad alanı (namespace)** bilgilerini bir araya getiren bir yapılandırma bilgisidir. **Context**, birden fazla Kubernetes kümesiyle çalışırken, hızlı bir şekilde geçiş yapmanı sağlar.

Context'in İçerisinde Bulunan Bilgiler

Bir context şu üç temel bileşenden oluşur:

1. **Cluster** → Bağlanılacak Kubernetes kümesi
2. **User (AuthInfo)** → Kimlik doğrulama için kullanılan kullanıcı bilgisi
3. **Namespace (Opsiyonel)** → Varsayılan olarak hangi namespace içinde çalışılacağı

Kubernetes kümesine bağlanmak için `kubectl config` komutlarını kullanabilirsin.

- **Mevcut context'leri listeleme:**

```
kubectl config get-contexts
```

- **Aktif context'i gösterme:**

```
kubectl config current-context
```

- **Başka bir context'e geçiş yapma:**

  Örneğin:

```
kubectl config use-context <context-adı>
```

```
kubectl config use-context minikube
```

Küme (Cluster) ve Node Yönetimi

- **Node'ları listeleme:**

```
kubectl get nodes
```

- **Node detaylarını görüntüleme:**

```
kubectl describe node <node-adı>
```

1. `kubectl cluster-info`

Cluster hakkında genel bilgi verir.

Bu komut, Kubernetes cluster'ının **kontrol düzlemi bileşenlerini (API server, scheduler, controller-manager vs.)** listeleyerek çalışıp çalışmadığını gösterir.

🔷 **Örnek Çıktı:**

```
Kubernetes control plane is running at https://127.0.0.1:6443
CoreDNS is running at https://127.0.0.1:6443/api/v1/namespaces/kube-syste
m/services/kube-dns
```

🔷 **Hata ile karşılaşırsanız:**

Cluster'ın çalıştığını doğrulamak için `minikube start` veya `kubectl config current-context` gibi komutları kullanabilirsiniz.

---

2. `kubectl get nodes`

Cluster'daki worker ve master node'ları listeler.

Bir Kubernetes cluster'ı **node'lardan** oluşur (master ve worker). Bu komut, mevcut node'ların listesini ve durumlarını gösterir.

🔷 **Örnek Çıktı:**

```
NAME       STATUS   ROLES    AGE    VERSION
node-1     Ready    master   10d    v1.25.3
worker-1   Ready    worker   10d    v1.25.3
```

🔷 **Önemli Alanlar:**

- **NAME:** Node'un adı
- **STATUS:** `Ready`, `NotReady`, `Unknown`

- **ROLES:** `master`, `worker`
- **AGE:** Node'un yaşı
- **VERSION:** Kubernetes sürümü

---

3. `kubectl get pods -n ms`

Belirtilen namespace içindeki pod'ları listeler.

Bu komut, **"ms"** adındaki namespace içindeki **pod'ları** gösterir.

🔷 **Örnek Çıktı:**

```
NAME         READY   STATUS    RESTARTS   AGE
web-pod       1/1    Running   0          2d
api-pod      1/1     Running   1          2d
db-pod       1/1     Running   0          2d
```

---

4. `kubectl get pods -A`

Tüm namespace'lerdeki pod'ları listeler.

- `A` flag'i (`-all-namespaces`), **tüm namespace'lerdeki** pod'ları gösterir.

🔷 **Örnek Çıktı:**

```
NAMESPACE    NAME           READY   STATUS    RESTARTS   AGE
kube-system  coredns-abc12    1/1    Running   0          3d
default      web-app          1/1    Running   2          5h
ms           api-service      1/1    Running   1          10h
```

---

5. `kubectl get pods -A -o wide`

Pod'lar hakkında daha geniş bilgi sağlar.

Bu komut, pod'ların hangi node üzerinde çalıştığını, IP adreslerini ve görüntüleri (image) gösterir.

🔷 **Örnek Çıktı:**

```
NAMESPACE   NAME           READY  STATUS   NODE    IP         IMAGE
kube-system coredns-abc12  1/1    Running  node-1  10.0.0.2   coredns:v1.8.4
default     web-app        1/1    Running  node-2  10.0.0.3   nginx:latest
ms          api-service    1/1    Running  node-3  10.0.0.4   my-api:v2
```

🔷 **Önemli Alanlar:**

- **NODE:** Pod'un çalıştığı node
- **IP:** Pod'un IP adresi
- **IMAGE:** Kullanılan container imajı

---

6. kubectl get pods -A -o yaml

Tüm pod'ları detaylı olarak YAML formatında gösterir.

Bu komut, pod'ların **tüm konfigürasyonunu** (metadata, spec, status gibi) YAML formatında döker.

🔷 **Örnek Çıktı (Kısaltılmış):**

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
  namespace: default
spec:
  containers:
  - name: web
    image: nginx:latest
status:
  phase: Running
```

---

7. `kubectl get pods -A -o json`

Tüm pod'ları JSON formatında gösterir.

Bu komut, `-o yaml` gibi detaylı bilgi verir ancak **JSON formatında** döker.

🔷 **Örnek Çıktı (Kısaltılmış):**

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "web-app",
    "namespace": "default"
  },
  "spec": {
    "containers": [
      {
        "name": "web",
        "image": "nginx:latest"
      }
    ]
  },
  "status": {
    "phase": "Running"
  }
}
```

---

8. `kubectl get pods -A -o jsonpath`

JSON formatında belirli bir değeri çekmek için kullanılır.

JSONPath, JSON içindeki belirli alanları çıkarmak için kullanılır.

🔷 **Örnek Kullanım:**

```
kubectl get pods -A -o jsonpath="{.items[*].metadata.name}"
```

🔷 **Örnek Çıktı:**

```
web-app api-service db-pod
```

Bu komut, **tüm pod'ların sadece isimlerini** listeler.

---

9. kubectl get pods -A -o go-template

Pod bilgilerini Go template kullanarak biçimlendirir.

Go template formatı ile verileri özelleştirebilirsiniz.

🔷 **Örnek Kullanım:**

```
kubectl get pods -A -o go-template='{{range .items}}{{.metadata.name}}:{{.st
atus.phase}}{{"\n"}}{{end}}'
```

🔷 **Örnek Çıktı:**

```
web-app:Running
api-service:Running
db-pod:Pending
```

Bu, **pod isimlerini ve durumlarını yan yana gösterir**.

---

10. kubectl get pods -A -o custom-columns

Kolonları özelleştirerek çıktı alır.

Bu komut, istediğiniz alanları özel bir tablo şeklinde formatlamanızı sağlar.

🔷 **Örnek Kullanım:**

```
kubectl get pods -A -o custom-columns="POD NAME:.metadata.name, STATU
S:.status.phase"
```

🔷 **Örnek Çıktı:**

```
POD NAME     STATUS
web-app      Running
api-service  Running
db-pod       Pending
```

Bu, sadece **pod ismini ve durumunu** gösterir.

Pod Nedir?

- **Kubernetes'te oluşturulup yönetilebilen en küçük objedir.**
- **Bir veya birden fazla container içerebilir, ancak çoğu durumda tek bir container barındırır.**
- **Her pod'un benzersiz bir kimliği (UID) vardır.**
- **Aynı pod içindeki container'lar aynı node üzerinde çalışır ve birbirleriyle `localhost` üzerinden haberleşir.**

1. `kubectl run firstpod --image=nginx --restart=Never`

Yeni bir pod oluşturur.

---

2. `kubectl describe pods firstpod`

Pod hakkında detaylı bilgi verir.

---

3. `kubectl logs firstpod`

Pod'un loglarını gösterir.

---

4. `kubectl logs -f firstpod`

Pod'un loglarını sürekli olarak takip eder.

---

5. `kubectl exec firstpod -- hostname`

Pod içinde `hostname` komutunu çalıştırır.

---

6. `kubectl exec -it firstpod -- /bin/sh`

Pod içinde interaktif bir shell başlatır.

---

7. `kubectl delete pods firstpod`

Pod'u siler.

YAML Nedir?

YAML (Yet Another Markup Language / YAML Ain't Markup Language), yapılandırma dosyaları ve veri formatları için kullanılan, okunabilirliği yüksek bir veri serileştirme dilidir. **Kubernetes'te kaynakları tanımlamak için yaygın olarak kullanılır.**

Kubernetes YAML Dosyasında Temel Bileşenler

1. `apiVersion:`

Kaynağın hangi API sürümüne ait olduğunu belirtir.

Örneğin:

```
apiVersion: v1
```

- **v1** → Temel kaynaklar (Pod, Service vb.)
- **apps/v1** → Deployment gibi kaynaklar için

---

2. `kind:`

Tanımlanan Kubernetes objesinin türünü belirler.

Örneğin:

```
kind: Pod
```

- **Pod**, **Service**, **Deployment**, **ConfigMap** gibi kaynaklar olabilir.

---

3. `metadata:`

Kaynak hakkında bilgi içerir (isim, etiketler, namespace vb.).

Örneğin:

```
metadata:
  name: my-pod
  labels:
    app: nginx
```

- **name:** Kaynağın adı
- **labels:** Kaynağı gruplamak için kullanılan anahtar-değer çiftleri

---

4. `spec:`

Kaynağın nasıl çalışacağını belirleyen yapılandırma ayarlarıdır.

Örneğin:

```
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
```

- **containers:** Pod içindeki container'ları tanımlar
- **image:** Kullanılacak container imajı

Örnek Kubernetes Pod YAML Dosyası

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: my-pod
  labels:
    app: nginx
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
      ports:
        - containerPort: 80
```

Bu YAML dosyası, **"my-pod"** adlı bir **nginx** pod'u oluşturur ve 80 numaralı portu açar.

1. `kubectl apply -f pod.yaml`

YAML dosyasındaki konfigürasyona göre bir pod oluşturur veya günceller.

---

2. `kubectl describe pods firstpod`

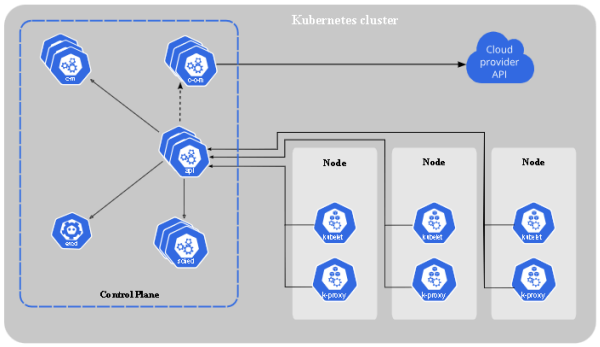
Belirtilen pod hakkında detaylı bilgi verir.

---

3. `kubectl edit pods firstpod`

Pod'un YAML konfigürasyonunu düzenlemek için bir editör açar.

Pod Yaşam Döngüsü



Bu görsel, bir **Kubernetes cluster'ının** temel bileşenlerini ve podların yaşam döngüsünü anlamak için kullanılabilir. **Pod yaşam döngüsünü** açıklamak için aşağıdaki adımları kullanabiliriz:

---

1. Pod’un Oluşturulması (Pending)

- **Pod tanımlanır ve API Server’a gönderilir.**
- `kubectl apply -f pod.yaml` gibi bir komut ile pod’un tanımı Kubernetes API Server’a gider.
- API Server, bu isteği **etcd** veritabanına kaydeder ve ilgili **scheduler'a** yönlendirir.

---

2. Pod’un Planlanması (Scheduled)

- **Kubernetes Scheduler, pod’un çalışacağı uygun bir node belirler.**
- Scheduler, **CPU, bellek ve node sağlık durumunu** değerlendirerek pod’u bir node’a yerleştirir.

---

3. Pod’un Çalıştırılması (Running)

- **Node üzerindeki Kubelet, pod’un çalıştırılmasını sağlar.**
- Kubelet, pod içindeki container'ları çalıştırmak için container runtime (Docker, containerd, CRI-O) ile iletişime geçer.

---

4. Pod’un Kullanımı ve Yönetimi

- Pod **uygulama trafiğini** alır ve işler.
- **Kubelet**, pod'un sağlığını sürekli olarak izler.
- **Kube-proxy**, pod'lar arasındaki ve dış dünyaya olan ağ trafiğini yönetir.

---

5. Pod'un Sonlanması (Terminating)

- Pod silindiğinde (`kubectl delete pod <pod-name>`), **Graceful Shutdown** süreci başlar.
- Kubelet, pod içindeki tüm container'ları durdurur ve scheduler'a bildirir.
- API Server, pod'un silindiğini **etcd** içinde günceller.

Kubernetes'te Çoklu Container Pod Nedir?

Kubernetes'te bir **Pod**, **bir veya birden fazla container içerebilen en küçük dağıtım birimidir**.

🔷 **Tek Container Pod** → Genellikle her Pod, **tek bir container** içerir ve uygulamanın bağımsız çalışmasını sağlar.

🔷 **Çoklu Container Pod** → Bir Pod içinde **birden fazla container** çalıştırılabilir. Bu container'lar aynı **network (localhost üzerinden haberleşir), depolama (volumes paylaşır)** ve yaşam döngüsünü paylaşır.

📌 Neden Çoklu Container Pod Kullanılır?

Çoklu container bir Pod içinde çalıştırılmasının başlıca **3 ana nedeni** vardır:

1️⃣ Sidecar Pattern (Yan Arabirim Container'ı)

- Ana uygulamanın yanında **log toplama, proxy veya monitoring** gibi işlemler için ekstra container kullanılır.
- Örnek: **Bir web sunucusu (Nginx) + Bir log toplama container'ı (Fluentd).**

2️⃣ Adapter Pattern (Dönüştürücü Container'lar)

- Ana container'ın **girdi veya çıktısını değiştirmek için** ek bir container kullanılır.
- Örnek: **Bir uygulama container'ı + Log formatını değiştiren bir adapter container.**

3️⃣ Ambassador Pattern (Proxy / Load Balancer)

- Ana container'ın dış sistemlerle bağlantısını yöneten bir ara katman olarak kullanılır.
- Örnek: **Bir veri tabanına bağlanan uygulama container'ı + Proxy container.**

🚀 Çoklu Container Pod Kullanımı

📌 Pod'u çalıştırmak için:

```
kubectl apply -f multi-container-pod.yaml
```

📌 Pod'un loglarını görmek için:

```
kubectl logs multi-container-pod -c nginx-container
kubectl logs multi-container-pod -c sidecar-container
```

📌 Pod'un içine girip test yapmak için:

```
kubectl exec -it multi-container-pod -c nginx-container -- /bin/sh
```

⚡ Çoklu Container Pod Kullanırken Dikkat Edilmesi Gerekenler:

❗ Container'lar **bağımsız ölçeklenemez**, çünkü Pod'un tamamı ölçeklenir.

❗ **CPU ve RAM paylaşımı** yapıldığı için kaynak yönetimi kritik önem taşır.

❗ Eğer bağımsız ölçekleme gerekiyorsa, container'ları **ayrı Pod'lar olarak dağıtmak daha mantıklıdır**.

📌 Init Container Nedir?

Kubernetes'te **Init Container**, bir Pod içindeki **ana (main) container başlamadan önce çalışan özel bir konteynerdir**.

Bunu, bir uygulamayı çalıştırmadan önce yapılması gereken **hazırlık adımları** gibi düşünebilirsin.

📌 Neden Init Container Kullanılır?

Bir uygulama başlamadan önce bazen **ön hazırlık işlemleri** yapmamız gerekir. Örneğin:

1. **Gerekli dosyaları veya konfigürasyonu indirmek**
2. **Veritabanının hazır olup olmadığını kontrol etmek**
3. **Güvenlik veya erişim izinlerini ayarlamak**
4. **Başka bir servisin ayağa kalkmasını beklemek**

Eğer bu işlemleri **ana container içinde yaparsan**, uygulama gereksiz yükle başlar ve hata yönetimi zor olur.

Bu yüzden **Init Container** kullanarak, **ön hazırlıkları yaparız ve ancak her şey hazır olduğunda ana container başlar**.

📌 Init Container Kullanımına Örnek YAML

Şimdi **Init Container'ın nasıl tanımlandığını** görelim:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  initContainers:
  - name: init-wait-for-db
    image: busybox
    command: ['sh', '-c', 'until nc -z db-service 5432; do echo waiting for db; sl
eep 2; done']
  containers:
  - name: main-app
    image: my-app:latest
```

```
    ports:
    - containerPort: 8080
```

📌 Ne Oluyor?

1. **Init Container (`init-wait-for-db`) çalışıyor ve `db-service` adlı veritabanının 5432 portunun açılmasını bekliyor.**
2. **Veritabanı hazır olduğunda** Init Container bitiyor.
3. **Ana uygulama container'ı (`main-app`) başlıyor.**

Eğer veritabanı hazır değilse, **ana container başlamaz ve hata almaz.** Böylece hata yönetimi kolaylaşır.

✅ Label (Etiketler):

Kubernetes nesnelerine eklenen **key-value (anahtar-değer)** çiftleridir. Nesneleri gruplamak ve yönetmek için kullanılır.

Örnek:

```
metadata:
  labels:
    app: myapp
    env: production
```

✅ Selector (Seçiciler):

Belirli etiketlere sahip nesneleri seçmek için kullanılır.

Örnek: Bir **Service** belirli label'a sahip Pod'lara yönlendirme yapar:

```
spec:
  selector:
    app: myapp
```

Selector Türleri:

🔷 **Equality-based:** `key=value` veya `key!=value`

🔷 **Set-based:** `key in (value1, value2)` veya `key notin (value1, value2)`

Örnek:

```
spec:
  selector:
    matchExpressions:
      - key: env
        operator: In
        values: [production, staging]
```

📌 **Kullanım Alanları:**

- **Servisler**, doğru Pod'lara trafik yönlendirmek için.
- **Deployment'lar**, belirli Pod'ları yönetmek için.
- **kubectl get pods --selector app=myapp**, belirli Pod'ları listelemek için.

1️⃣ Pod'ları Etikete Göre Listeleme

🔷 **Belirli etiketlere sahip Pod'ları getir:**

```
kubectl get pods -l "app=firstapp, tier=backend"
```

🔷 **Tüm Pod'ların etiketlerini de göster:**

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

🔷 **Belirli bir etikete sahip olmayanları listele:**

```
kubectl get pods -l 'app!=firstapp'
```

🔷 **Bir etikete sahip olan tüm Pod'ları göster:**

```
kubectl get pods -l 'app'
```

2️⃣ Belirli Birkaç Değerden Birine Sahip Pod'ları Listeleme

```
kubectl get pods -l 'app in (firstapp, secondapp)'
```

Bu komut, **app=firstapp** veya **app=secondapp** etiketine sahip Pod'ları listeler.

3️⃣ Belirli Değerlere Sahip Olmayan Pod'ları Listeleme

```
kubectl get pods -l 'app notin (firstapp, secondapp)'
```

Bu komut, **app etiketi firstapp veya secondapp olmayan** Pod'ları getirir.

Mevcut Bir Etiketi Değiştirme (Overwrite)

Eğer bir **Pod** veya başka bir Kubernetes nesnesinde var olan bir etiketi değiştirmek istiyorsan `--overwrite` bayrağını kullanabilirsin.

Örnek: Mevcut Etiketi Güncelleme

```
kubectl label pods pod9 team=team3 --overwrite
```

✅ Eğer `my-pod` zaten **env=production** etiketine sahipse, **env=staging** olarak değiştirir.

Kubernetes Annotation (Açıklamalar) Nedir?

Annotation, Kubernetes nesnelerine **metadata (ek bilgi)** eklemek için kullanılan key-value (anahtar-değer) çiftleridir. **Labels (Etiketler)** gibi çalışır, ancak **nesneleri seçmek veya filtrelemek için kullanılamaz**. Daha çok **insanlar veya sistemler için ek bilgi eklemek** amacıyla kullanılır.

🔷 Annotation Kullanımı (Pod Örneği)

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: myapp
  annotations:
    description: "Bu pod backend servisine ait."
    owner: "Mustafa"
    git-repo: "https://github.com/myrepo"
    checksum/config: "123456abc"
spec:
  containers:
    - name: nginx
      image: nginx
```

Bir Nesnenin Annotation'larını Görüntüleme

```
kubectl get pod example-pod --show-labels
kubectl describe pod example-pod
```

✅ `describe` komutu ile Pod'un tüm annotation bilgilerini görebilirsin.

Kubernetes Namespace Nedir?

Namespace, Kubernetes'te kaynakları **mantıksal olarak ayırmak ve izole etmek** için kullanılan bir yapıdır.

Varsayılan Namespace'ler

Kubernetes'te **4 temel namespace** vardır:

1. **default** → Varsayılan namespace (Eğer belirtmezsen buraya eklenir)

2. **kube-system** → Kubernetes'in kendi sistem bileşenleri (DNS, scheduler, controller vs.)
3. **kube-public** → Herkes tarafından görülebilen genel bilgiler (ConfigMap vb.)
4. **kube-node-lease** → Node health check mekanizması için kullanılır.

🔷 CLI ile Namespace Yönetimi

1️⃣ Namespace'leri Listeleme

```
kubectl get namespaces
```

2️⃣ Yeni Namespace Oluşturma

```
kubectl create namespace my-namespace
```

3️⃣ Namespace İçindeki Kaynakları Listeleme

```
kubectl get pods -n my-namespace
```

4️⃣ Default Namespace'i Güncellemek

```
kubectl config set-context --current --namespace=my-namespace
```

5️⃣ Namespace Silme

```
kubectl delete namespace my-namespace
```

🔷 Namespace Kullanarak Pod Oluşturma

Eğer bir nesneyi belirli bir namespace içinde oluşturmak istiyorsan, YAML içinde belirtmelisin:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  namespace: my-namespace
spec:
  containers:
    - name: nginx
      image: nginx
```

✅ Bu Pod `my-namespace` içinde çalışacaktır.

Kubernetes Deployment Nedir?

Deployment, Kubernetes'te **pod'ları yönetmek ve ölçeklendirmek** için kullanılan en yaygın kaynaktır.

✔️ **Pod'ları oluşturur, günceller ve siler.**

✔️ **İstediğin sayıda pod'un çalışmasını garanti eder (ReplicaSet).**

✔️ **Canlıya sorunsuz güncelleme yapmayı sağlar (Rolling Update).**

✔️ **Crash eden pod'ları otomatik yeniden başlatır.**

Deployment YAML Örneği

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    app: myapp
spec:
  replicas: 3  # 3 pod çalıştır
  selector:
    matchLabels:
```

```
    app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: my-container
        image: nginx
        ports:
        - containerPort: 80
```

📌 **Bu ne yapar?**

- **3 pod** çalıştırır. (replicas: 3)
- **Pod'ları** `app=myapp` **etiketiyle eşleştirir.**
- **Container olarak** `nginx` **çalıştırır.**
- **80 portunu açar.**

Deployment'ı Oluşturma

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

✅ **YAML dosyasındaki deployment'ı cluster'a ekler.**

Deployment'ları Listeleme

```
kubectl get deployments
```

✅ **Mevcut deployment'ları listeler.**

Deployment'ı Güncelleme (Rolling Update)

```
kubectl set image deployment/my-deployment my-container=nginx:latest
```

✅ **Tüm pod’ları yeni image ile günceller.**

✅ **Eski pod’ları sırasıyla kapatır, yenilerini açar.**

Ölçeklendirme (Scaling)

```
kubectl scale deployment my-deployment --replicas=5
```

✅ **Pod sayısını 3’ten 5’e çıkarır.**

Rollback (Önceki Versiyona Geri Dönme)

```
kubectl rollout undo deployment my-deployment
```

✅ **Hatalı güncellemeyi geri alır.**

Deployment’ı Silme

```
kubectl delete deployment my-deployment
```

✅ **Deployment’ı ve ilgili tüm pod’ları siler.**

Kubernetes Deployment Strategy (Strateji) Nedir?

Deployment Strategy (Dağıtım Stratejisi), Kubernetes’te uygulamalarını **nasıl güncelleyeceğini** belirleyen mekanizmadır.

✔️ **Yeni versiyona kesintisiz geçiş (Rolling Update)**

✔️ **Eski versiyonu tamamen kapatıp yeni versiyonu açma (Recreate)**

✔️ **A/B testi, blue-green veya canary gibi özel stratejiler**

🔷 Kubernetes’te 2 Ana Deployment Stratejisi Vardır:

1️⃣ **RollingUpdate (Varsayılan, Kesintisiz Güncelleme)**

2️⃣ **Recreate (Eskiyi Kapat, Yeniyi Aç)**

1️⃣ RollingUpdate (Kesintisiz Güncelleme)

🔷 **Eski pod'ları sırayla kapatıp yenilerini açar.**

🔷 **Hizmet kesintisi olmaz (Zero Downtime).**

🔷 **Deployment’ı güncellerken belirli sayıda pod’un her zaman çalışmasını sağlar.**

Örnek YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1  # Aynı anda en fazla 1 pod down olabilir
      maxSurge: 1        # Güncelleme sırasında en fazla 1 yeni pod açılabilir
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: my-container
          image: nginx:latest
```

✅ **Nasıl Çalışır?**

- **maxUnavailable: 1** → Güncelleme sırasında **en fazla 1 pod kapanabilir.**
- **maxSurge: 1** → Güncelleme sırasında **en fazla 1 yeni pod açılabilir.**
- **Pod'lar sırayla güncellenir, downtime olmaz.**

2️⃣ Recreate (Eskiyi Kapat, Yeniyi Aç)

🔷 **Önce tüm eski pod'ları siler, sonra yeni pod'ları açar.**

🔷 **Downtime (kesinti) olur!**

🔷 **Durum bilgisi tutan (stateful) uygulamalar için risklidir.**

Örnek YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: my-container
          image: nginx:latest
```

✅ **Nasıl Çalışır?**

1️⃣ **Önce tüm mevcut pod'ları kapatır.**

2️⃣ **Sonra yeni versiyonu çalıştırır.**

3️⃣ **Bir süreliğine uygulama erişilemez olur (Downtime yaşanır).**

Kubernetes Ağ (Networking) Altyapısı

Kubernetes'te **ağ iletişimi**, cluster içindeki **Pod'lar, Service'ler ve dış dünya** arasındaki bağlantıları düzenleyen kritik bir bileşendir. Kubernetes, **ağ modelini merkeziyetsiz ve düz bir yapıda tasarlamıştır**, yani her Pod kendi IP adresine sahiptir ve birbirleriyle doğrudan iletişim kurabilir.

1️⃣ Kubernetes Ağ Modeli

Kubernetes, aşağıdaki ağ ilkelerine dayanır:

1. **Her Pod kendi IP adresine sahiptir.**
   - Pod'lar aynı node içinde de olsa farklı node'larda da olsa **birbirleriyle doğrudan iletişim kurabilirler.**
2. **Pod'lar arasındaki iletişim NAT gerektirmez.**
   - Pod'lar arasında ağ izolasyonu yoktur, ancak NetworkPolicy ile sınırlamalar eklenebilir.
3. **Node'lar ve Pod'lar birbirleriyle doğrudan konuşabilir.**
   - Yani bir Node üzerindeki Pod, başka bir Node üzerindeki Pod ile özel bir yapılandırmaya gerek kalmadan konuşabilir.
4. **Kubernetes Service'ler, Pod'ları soyutlamak için bir ağ soyutlaması sağlar.**
   - Pod'lar değişken olduğu için (ölçeklenme, yeniden başlama gibi) Service'ler sabit bir erişim noktası oluşturur.

2️⃣ Kubernetes Ağ Bileşenleri

Kubernetes'te ağ iletişimini sağlayan temel bileşenler şunlardır:

🔷 1. Pod-to-Pod İletişimi

- **Her Pod kendi IP'sine sahiptir ve bu IP ile başka Pod'larla doğrudan iletişim kurabilir.**
- Pod'lar arasındaki iletişim **Node'un kendisi veya bir overlay network (örn. Flannel, Calico, Cilium) aracılığıyla sağlanır.**
- Pod'ların ölçeklenmesi durumunda, bir Service ile erişim stabil hale getirilir.

🔷 2. Service-to-Pod İletişimi

Pod'lar **dinamik olarak ölçeklenebilir ve değişebilir**. Bir Pod'un IP'si değişirse, ona doğrudan erişim zorlaşır.

Service'ler, **Pod'ları soyutlayarak** onlara sabit bir erişim noktası sunar.

Bu sayede dış dünyadan veya diğer Pod'lardan gelen istekleri yönlendirebilir.

Service türleri:

- **ClusterIP** (Default) → Sadece Cluster içinden erişilebilir.
- **NodePort** → Cluster dışından **Node'un IP'si ve belirlenen port ile** erişilebilir.
- **LoadBalancer** → Bulut ortamında bir **load balancer** üzerinden dış erişim sağlar.
- **ExternalName** → DNS üzerinden bir dış kaynağa yönlendirme yapar.

🔷 3. Node-to-Pod İletişimi

- Kubernetes'te Node'lar, Pod'ları çalıştıran fiziksel veya sanal makineler (VM) olarak düşünülebilir.
- Node'lar üzerinde çalışan **Kubelet**, Pod'ların çalışmasını kontrol eder.
- **Kube-Proxy**, **Servislerin arkasındaki Pod'lara trafiği yönlendirmek için** çalışır.

🔷 4. Pod-to-External İletişim (Dış Dünya)

Pod'ların dış dünyayla iletişimi **Service'ler veya Ingress ile sağlanır**:

a) Service ile Dış Dünya İletişimi

- `NodePort` ve `LoadBalancer` kullanarak **Pod'lara dış dünyadan erişim** sağlanabilir.

b) Ingress ile HTTP/HTTPS Trafik Yönetimi

- **Ingress**, HTTP/HTTPS isteklerini **farklı Service'lere yönlendirmek** için kullanılır.
- **Örnek:**

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

  - `example.com` adresine gelen istekler **frontend-service** adlı Service'e yönlendirilir.

3️⃣ Kubernetes Ağ Çözümleri (CNI - Container Network Interface)

Kubernetes ağ altyapısını yönetmek için farklı **CNI (Container Network Interface) çözümleri** kullanılır:

Ağ Çözümü	Özellikleri
Flannel	Basit ve hızlıdır, Pod'lar arası ağ iletişimini sağlar.

Calico	Ağ politikalarını (Network Policies) destekler, güvenlik sağlar.
Cilium	eBPF tabanlı, performans odaklıdır, ileri seviye güvenlik sunar.
Weave	Basit ve otomatik mesh ağ desteği sunar.

- Eğer Kubernetes Cluster'ında **NetworkPolicy** kullanarak güvenlik sağlamak istiyorsan, **Calico veya Cilium** gibi çözümleri kullanman gerekir.

4️⃣ Kubernetes Ağ Politikaları (Network Policies)

Normalde Kubernetes'te **tüm Pod'lar birbirine erişebilir**, ancak **NetworkPolicy** ile trafiği sınırlayabilirsin.

Örnek: **Sadece belirli bir etikete sahip Pod'lardan gelen trafiğe izin ver**

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
```

Bu politika, **backend Pod'larına sadece frontend Pod'larından gelen trafiğe izin verir**.

Kubernetes Service (Hizmet) Nedir?

Kubernetes'te **Pod'lar dinamik olarak oluşturulur ve silinir**, bu yüzden **doğrudan bir Pod'un IP'sine güvenmek mümkün değildir**. İşte bu noktada **Service** devreye girer.

Service, **Pod'ları sabit bir IP ve DNS ile erişilebilir hale getiren bir ağ soyutlamasıdır**.

Bir **Service** oluşturduğunda, ilgili Pod'lara **otomatik yük dengeleme (load balancing) ve sabit erişim sağlar**.

1️⃣ Neden Service'e İhtiyacımız Var?

- **Pod'ların IP adresi değişkendir** → Yeni bir Pod oluşturulursa farklı bir IP alır.
- **Pod'ları sabit bir adrese bağlamak gerekir** → Service, bir DNS ve sabit IP sağlar.
- **Yük dengeleme gerekir** → Service, birden fazla Pod arasında istekleri dağıtır.
- **Dış dünyadan erişim sağlamak gerekir** → Service, Pod'ları dış dünyaya açabilir.

2️⃣ Kubernetes Service Türleri

Kubernetes'te dört farklı **Service türü** vardır:

Service Türü	Açıklama	Kullanım Senaryosu
ClusterIP (Default)	Yalnızca cluster içinden erişilebilir.	Mikroservisler arası iletişim.
NodePort	Node'un belirli bir portu üzerinden dış erişim sağlar.	Basit dış erişim.
LoadBalancer	Bulut ortamında otomatik yük dengeleyici oluşturur.	Bulut servisleri (AWS, GCP, Azure).
ExternalName	Harici bir DNS adına yönlendirme yapar.	Kubernetes dışındaki servisleri kullanmak.

3️⃣ Service Türlerine Detaylı Bakış

📌 a) ClusterIP (Varsayılan)

- Pod’ları **sadece cluster içinden** erişilebilir hale getirir.
- Varsayılan Service türüdür.
- **Pod'lar arasındaki mikroservis iletişimi** için kullanılır.

📌 Örnek ClusterIP Service Tanımı

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

📌 Nasıl Kullanılır?

```
kubectl get service my-service
```

- Bu Service, `app=my-app` etiketine sahip Pod’lara trafik yönlendirir.
- Pod’lar `my-service` DNS adıyla bu servise ulaşabilir.

📌 b) NodePort

- Cluster dışından erişime izin verir.
- **Her Node'un belirlenen bir portunu açar (30000-32767 arası).**
- Service’in bağlı olduğu tüm Node’lardan erişilebilir.

📌 Örnek NodePort Service Tanımı

```
apiVersion: v1
kind: Service
metadata:
  name: my-nodeport-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 30007
```

📌 **Nasıl Kullanılır?**

```
kubectl get service my-nodeport-service
```

- Service, her **Node'un IP'sinden** şu şekilde erişilebilir:

  ```
  http://<Node-IP>:30007
  ```

📌 c) LoadBalancer

- **Bulut ortamında (AWS, GCP, Azure) otomatik bir yük dengeleyici oluşturur.**
- **Dış dünyadan erişim sağlar.**
- NodePort'un bir üst seviyesidir.

📌 **Örnek LoadBalancer Service Tanımı**

```
apiVersion: v1
kind: Service
metadata:
  name: my-loadbalancer
```

```
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

📌 **Nasıl Kullanılır?**

```
kubectl get service my-loadbalancer
```

- **Bulut sağlayıcısı**, otomatik olarak bir dış IP atar.
- Eğer **Minikube gibi bir lokal cluster** kullanıyorsan:komutu ile lokal LoadBalancer IP'sine yönlendirme yapabilirsin.

  ```
  minikube service my-loadbalancer
  ```

📌 d) ExternalName

- Kubernetes dışındaki bir **DNS adına yönlendirme yapar.**
- `kubectl get service` ile bakıldığında **gerçek bir IP yerine DNS adı görünür.**

📌 **Örnek ExternalName Service Tanımı**

```
apiVersion: v1
kind: Service
metadata:
  name: my-external-service
spec:
  type: ExternalName
  externalName: example.com
```

📌 **Nasıl Kullanılır?**

- Cluster içindeki Pod'lar, `my-external-service` DNS adıyla doğrudan `example.com`'a yönlendirilir.

4️⃣ Service ve Selector Mantığı

Bir Service, **Pod'ları etiketlere (labels) göre bulur** ve trafiği onlara yönlendirir.

📌 **Service Selector Mantığı:**

```
spec:
  selector:
    app: my-app
```

- **Bu Service, `app=my-app` etiketine sahip tüm Pod'lara trafik yönlendirir.**

Pod Tanımı:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-app
spec:
  containers:
  - name: my-container
    image: nginx
```

- Bu Pod, `app=my-app` etiketi taşıdığı için yukarıdaki Service'e dahil olur.

📌 **Service'lerin bağlı olduğu Pod'ları görmek için:**

```
kubectl get endpoints my-service
```

Bu komut, Service'in hangi **Pod IP'lerine yönlendirme yaptığını** gösterir.

5️⃣ Service DNS Kullanımı

Kubernetes **Service'lere otomatik bir DNS kaydı atar**.

📌 **Varsayılan DNS Formatı:**

```
http://<service-name>.<namespace>.svc.cluster.local
```

📌 **Örnek:**

- `my-service` adlı bir Service **default namespace** içinde çalışıyorsa erişilebilir.

  ```
  http://my-service.default.svc.cluster.local
  ```

6️⃣ Service Kullanım Senaryoları

Senaryo	Hangi Service Türü?
Pod'lar arasında iletişim	ClusterIP
Lokal geliştirme ortamında dış erişim	NodePort
Bulutta çalışan bir web uygulaması	LoadBalancer
Dış DNS adresine yönlendirme	ExternalName

7️⃣ Service vs Ingress

Service, **Pod'ları yönetmek ve yönlendirmek için** kullanılır.

Ingress, **HTTP/HTTPS trafik yönetimi yapar ve Service'lere yönlendirir.**

Özellik	Service	Ingress
Amaç	Pod'lara yönlendirme	HTTP yönlendirme
L7 (Application Layer) Routing	❌	✅
Load Balancer Yönetimi	✅ (LoadBalancer tipi ile)	✅ (Ingress Controller ile)
Güvenlik & TLS Desteği	❌	✅

- **Service**, **Pod'lar arasındaki iletişimi yönetir.**
- **Ingress**, **dış dünyadan gelen HTTP/HTTPS trafiğini yönetir.**
- Genellikle **Ingress + ClusterIP Service kombinasyonu kullanılır**.

Liveness Probe (Canlılık Kontrolü) Nedir?

Kubernetes'te **Liveness Probe**, bir **container'ın hala sağlıklı (canlı) olup olmadığını kontrol eden mekanizmadır**.

Eğer bir **Liveness Probe başarısız olursa, Kubernetes o container'ı öldürüp tekrar başlatır**.

Liveness Probe, özellikle **kendini toparlayamayacak şekilde sıkışan (hang) veya çöken uygulamaları otomatik olarak yeniden başlatmak için** kullanılır.

1️⃣ Neden Liveness Probe Kullanmalıyız?

- **Container içindeki uygulama çökebilir, ama işlem hala çalışıyor olabilir.**
- **Uygulama kilitlenebilir ve hiçbir yanıt vermeyebilir.**
- **Otomatik kurtarma (self-healing) mekanizması sağlamak için.**

Örnek Senaryo:

- Bir API sunucusu var, ancak veri tabanı bağlantısı koptu ve hiçbir yanıt dönmüyor.
- Kubernetes bunu fark edip container'ı yeniden başlatabilir.

2️⃣ Liveness Probe Türleri

Kubernetes, bir container'ın canlı olup olmadığını kontrol etmek için üç farklı yöntem sunar:

Probe Türü	Açıklama
HTTP Probe	Belirli bir endpoint'e HTTP isteği yapar.
Command Probe	Container içinde bir komut çalıştırır.
TCP Probe	Belirli bir porta bağlantı açmayı dener.

3️⃣ Liveness Probe Kullanımı

📌 a) HTTP Liveness Probe

Container içindeki bir HTTP endpoint'i çağırır ve **200-399** arası bir yanıt dönerse sağlıklı kabul eder.

📌 **Örnek:** `/healthz` endpoint'i üzerinden kontrol eden bir Liveness Probe

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
  - name: my-app
    image: my-app:latest
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
```

📌 **Açıklama:**

- `httpGet` → Pod'un `/healthz` endpoint'ine **port 8080** üzerinden request atar.
- `initialDelaySeconds: 5` → **Başlangıçta 5 saniye bekler.**
- `periodSeconds: 10` → **Her 10 saniyede bir kontrol eder.**
- Eğer **3 ardışık başarısızlık** olursa Kubernetes container'ı **öldürüp yeniden başlatır.**

📌 b) Command Liveness Probe

Container içinde belirli bir komutu çalıştırır. Eğer **çıkış kodu** `0` **dönerse sağlıklı kabul edilir**, `0` dışında bir kod dönerse container yeniden başlatılır.

📌 **Örnek:** Container içinde `/app/healthcheck.sh` komutunu çalıştıran Liveness Probe

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-command
spec:
  containers:
  - name: my-app
    image: my-app:latest
    livenessProbe:
      exec:
        command:
        - /bin/sh
        - -c
        - cat /tmp/healthy
      initialDelaySeconds: 3
      periodSeconds: 5
```

📌 **Açıklama:**

- `/bin/sh -c "cat /tmp/healthy"` → Eğer `cat /tmp/healthy` dosyası varsa (`0` döndürür), container sağlıklı kabul edilir.
- Eğer dosya yoksa, probe başarısız olur ve **Kubernetes container'ı öldürüp yeniden başlatır.**

📌 c) TCP Liveness Probe

Belirli bir porta bağlantı açmayı dener. Eğer port dinleniyorsa sağlıklı kabul edilir.

📌 **Örnek:** Container'ın **8080 portuna TCP bağlantısı açmayı deneyen** Liveness Probe

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-tcp
spec:
```

```
containers:
- name: my-app
  image: my-app:latest
  livenessProbe:
    tcpSocket:
      port: 8080
    initialDelaySeconds: 5
    periodSeconds: 10
```

📌 **Açıklama:**

- Eğer **8080 portu cevap veriyorsa**, Pod sağlıklı kabul edilir.
- Eğer uygulama **portu dinlemeyi durdurursa**, Kubernetes container'ı yeniden başlatır.

4️⃣ Liveness Probe Parametreleri

Tüm probe türleri aşağıdaki ortak parametrelere sahiptir:

Parametre	Açıklama
`initialDelaySeconds`	İlk kontrolün başlamadan önce kaç saniye bekleyeceğini belirtir.
`periodSeconds`	Liveness Probe'un ne sıklıkla çalışacağını belirtir.
`timeoutSeconds`	Probe'un zaman aşımı süresi.
`successThreshold`	Bir başarısızlıktan sonra kaç başarılı probe sonucu alındığında sağlıklı sayılacağını belirler.
`failureThreshold`	Kaç başarısız probe sonucunda container'ın yeniden başlatılacağını belirler.

5️⃣ Liveness Probe vs Readiness Probe vs Startup Probe

Probe Türü	Amacı	Başarısız Olursa Ne Olur?
Liveness Probe	**Container sağlıklı mı?**	Kubernetes container'ı öldürüp yeniden başlatır.

Readiness Probe	**Container trafiğe hazır mı?**	Service, bu container'a trafik göndermeyi durdurur.
Startup Probe	**Container başlangıç aşamasını geçti mi?**	Liveness Probe devreye girene kadar bekler.

📌 **Genellikle Liveness + Readiness birlikte kullanılır**:

- Liveness Probe **uygulamanın tamamen çöktüğünü** kontrol eder.
- Readiness Probe **uygulama başlarken hazır olup olmadığını** kontrol eder.
- Startup Probe, **uygulamanın geç başlama durumlarını ele alır.**

6️⃣ Gerçek Hayatta Kullanım Senaryoları

Senaryo	Çözüm
Container çöktü ama kapanmadı (process kill olmadı)	**Liveness Probe ile yeniden başlatılır.**
API servisleri yoğun yük altında geç cevap veriyor	**Readiness Probe ile trafiğe açılmadan önce beklenir.**
Java/Python gibi uzun sürede açılan uygulamalar	**Startup Probe kullanılır, Liveness Probe hemen devreye girmez.**

📌 **Örnek: Liveness + Readiness + Startup Probe birlikte kullanım**

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 10

readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 5
```

```
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  initialDelaySeconds: 30
  failureThreshold: 10
  periodSeconds: 5
```

Kubernetes Resource Limits (Kaynak Sınırları)

Kubernetes'te **Pod veya Container'ın ne kadar CPU ve RAM kullanabileceğini sınırlar.**

İki türü vardır:

- **Requests:** Minimum garanti edilen kaynaklar.
- **Limits:** Maksimum kullanılabilecek kaynaklar.

📌 **Örnek Kullanım:**

```
resources:
  requests:
    memory: "128Mi"
    cpu: "250m"
  limits:
    memory: "256Mi"
    cpu: "500m"
```

- **CPU (millicores):** `500m = 0.5 CPU`
- **Memory (RAM):** `256Mi = 256MB`

📌 **Limit Aşılırsa Ne Olur?**

- **CPU aşılırsa:** Pod yavaşlar (throttle edilir).
- **Memory aşılırsa:** Pod **Öldürülür! (OOMKilled)**

📌 **Varsayılan Sınırları Belirleme (`LimitRange`):**

```
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
spec:
  limits:
  - default:
      cpu: "500m"
      memory: "512Mi"
    defaultRequest:
      cpu: "250m"
      memory: "256Mi"
    type: Container
```

📌 **Kaynak Kullanımını Kontrol Etmek İçin:**

```
kubectl top pod my-pod
```

Hard-Coded Nedir?

Hard-coded, değerlerin doğrudan kod içine gömülmesidir.

Bu, **esneklik kaybına ve güvenlik risklerine** yol açar.

📌 **Örnek (Kötü Uygulama):**

```
db_host = "192.168.1.100"  # Hard-coded IP
db_user = "admin"
db_password = "12345"
```

🔷 Neden Kötü?

❌ **Esnek değildir** → Değişiklik için kodu güncellemek gerekir.

❌ **Güvensizdir** → Şifreler açıkta olur.

❌ **Yönetimi zordur** → Farklı ortamlar için ayarları değiştirmek karmaşıktır.

✅ Alternatif Çözümler

1️⃣ **Ortamdaki değişkenleri kullan (`os.getenv()`)**
2️⃣
Konfigürasyon dosyası (JSON/YAML) kullan
3️⃣
Secrets yönetimi (Kubernetes Secret, AWS Secrets Manager)

📌 **Doğru Yaklaşım:**

```
import os
db_host = os.getenv("DB_HOST")  # Ortam değişkeninden al
```

🚀 **Özet: Hard-coded kullanma!** Esneklik ve güvenlik için **dışarıdan yönetilebilir yapı** kullan.

Kubernetes Volume (Depolama)

Kubernetes'te **Volume**, Pod'lar içinde **kalıcı veya geçici veri saklamak** için kullanılır.

✅ **Container yeniden başlasa bile veri kaybolmaz (bazı türlerde).**

✅ **Birden fazla container aynı Volume'a erişebilir.**

✅ **Veri, Pod veya Node ölçeklendirilirken korunabilir.**

1️⃣ Volume Türleri

Kubernetes'te farklı türde Volume'ler vardır. En çok kullanılanları:

Volume Türü	Açıklama

`emptyDir`	Pod çalıştığı sürece geçici veri saklar. Pod silinince veri kaybolur.
`hostPath`	Node'un diskindeki belirli bir dizini Pod'a bağlar.
`persistentVolumeClaim (PVC)`	Kalıcı depolama alanı sağlar (Pod silinse bile veri korunur).
`configMap`	Config dosyalarını Volume olarak bağlamak için.
`secret`	Hassas bilgileri dosya olarak bağlamak için.

2️⃣ Volume Kullanımı

📌 Geçici Volume (`emptyDir`)

Pod içinde geçici bir alan oluşturur. **Pod silinince veriler de silinir.**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: nginx
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: my-storage
  volumes:
    - name: my-storage
      emptyDir: {}  # Pod çalıştığı sürece geçici veri saklar
```

✅ **Pod içindeki `/usr/share/nginx/html` dizini, bu Volume'a bağlanır.**

❌ **Pod silindiğinde veri kaybolur.**

📌 Kalıcı Volume (`PersistentVolumeClaim`)

Pod silinse bile verileri saklamak için kullanılır.

🔷 **Önce PVC (Persistent Volume Claim) tanımlanır:**

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

🔷 **Sonra Pod içinde kullanılır:**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: nginx
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: my-storage
  volumes:
    - name: my-storage
      persistentVolumeClaim:
        claimName: my-pvc  # PVC'yi burada kullanıyoruz
```

✅ **Pod silinse bile veri korunur.**

📌 Node Diskine Bağlanan Volume (`hostPath`)

Bu Volume, **Node'un diskindeki belirli bir dizini** Pod'a bağlar.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: nginx
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: my-storage
  volumes:
    - name: my-storage
      hostPath:
        path: "/data"
```

✅ **Node'daki `/data` dizini, Pod içinde `/usr/share/nginx/html` olarak bağlanır.**

❌ **Pod başka bir node'a taşınırsa bu veri kaybolabilir.**

📌 Kubernetes Secret (Hassas Veriler)

Kubernetes **Secret**, hassas verileri güvenli bir şekilde saklamak için kullanılır.

✅ **Şifreler, API anahtarları ve TLS sertifikalarını saklamak için kullanılır.**

✅ **ConfigMap'e benzer, ancak daha güvenlidir (Base64 şifreleme kullanır).**

✅ **Pod içine çevresel değişken olarak veya dosya olarak eklenebilir.**

1️⃣ Secret Tanımlama

Secret oluştururken, verileri Base64 formatında şifrelemeliyiz.

🔷 **Önce Base64 ile veriyi şifreleyelim:**

```
echo -n "mypassword" | base64
```

Çıktı:

```
bXIwYXNzd29yZA==
```

🔷 **Sonra Secret YAML dosyası oluşturalım:**

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  password: bXIwYXNzd29yZA==  # Base64 şifrelenmiş veri
```

2️⃣ Secret Kullanımı

Secret'ı **iki farklı şekilde** kullanabiliriz:

1️⃣ **Çevresel değişken olarak**

2️⃣ **Dosya olarak Volume'a mount ederek**

📌 1️⃣ Secret'ı Çevresel Değişken Olarak Kullanma

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: nginx
      env:
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
```

```
            name: my-secret
            key: password
```

✅ **Container içinde `DB_PASSWORD` ortam değişkeni olarak tanımlanır.**

✅ **Uygulama bunu `DB_PASSWORD` ortam değişkeninden okuyabilir.**

📌 2️⃣ Secret'ı Dosya Olarak Bağlama

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: nginx
      volumeMounts:
        - name: secret-volume
          mountPath: "/etc/secret"
  volumes:
    - name: secret-volume
      secret:
        secretName: my-secret
```

✅ **Secret, Pod içinde `/etc/secret/password` dosyası olarak bulunur.**

✅ **Uygulama dosyadan okuyabilir (`cat /etc/secret/password`).**

Kubernetes ConfigMap Nedir?

ConfigMap, Kubernetes'te **uygulama yapılandırmalarını** yönetmek için kullanılan bir nesnedir. **Şifreler ve hassas veriler için değil, genel yapılandırma bilgileri için kullanılır.**

✅ **Şifre içermeyen yapılandırma verilerini saklar.**

✅ **Pod'lar, bu verileri çevresel değişken, komut satırı argümanı veya dosya olarak kullanabilir.**

✅ **Uygulama kodundan bağımsız olarak yapılandırmayı değiştirmeye izin verir.**

📌 ConfigMap vs Secret

Özellik	ConfigMap	Secret
Amaç	Uygulama yapılandırmalarını saklamak	Hassas verileri saklamak (şifreler, API anahtarları)
Şifreleme	Düz metin olarak saklanır	Base64 ile şifrelenir
Kullanım Alanı	Bağlantı adresleri, yapılandırma değerleri, port bilgileri	Şifreler, token'lar, sertifikalar
Nasıl Kullanılır?	Çevresel değişken, dosya veya argüman olarak	Çevresel değişken veya dosya olarak

---

📌 ConfigMap Nasıl Kullanılır?

1️⃣ ConfigMap Oluşturma

ConfigMap **üç farklı şekilde** oluşturulabilir:

1. YAML dosyası ile
2. Komut satırı (`kubectl create configmap`) ile
3. Dosya içeriğinden

📌 1️⃣ YAML ile ConfigMap Tanımlama

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  database_url: "postgres://user:pass@db:5432/app"
```

```
  log_level: "debug"
  max_connections: "100"
```

✅ `database_url`, `log_level` ve `max_connections` **isimli konfigürasyonları tutar.**

📌 2️⃣ Kubectl ile ConfigMap Oluşturma

Komut satırından ConfigMap oluşturmak için:

```
kubectl create configmap my-config --from-literal=database_url="postgres://
user:pass@db:5432/app" --from-literal=log_level="debug"
```

✅ **Bu komut,** `database_url` **ve** `log_level` **adında iki değer içeren bir ConfigMap oluşturur.**

📌 3️⃣ Bir Dosyadan ConfigMap Oluşturma

Eğer bir yapılandırma dosyanız varsa, doğrudan bu dosyayı kullanabilirsiniz.

🔷 **Önce bir dosya oluştur:**

```
echo "database_url=postgres://user:pass@db:5432/app" > app-config.proper
ties
```

🔷 **Sonra ConfigMap'i oluştur:**

```
kubectl create configmap my-config --from-file=app-config.properties
```

✅ **Dosyanın içeriği ConfigMap içine kaydedilir.**

Node Affinity (Düğüm Bağımlılığı)

Node Affinity, bir Pod'un belirli niteliklere sahip düğümlerde çalışmasını sağlamak için kullanılır. Bu, özellikle belirli donanım özelliklerine, bölgelere (regions), veya düğüm etiketlerine (labels) göre Pod'ları yönlendirmek için kullanılır.

Çalışma Mantığı

Node Affinity, `nodeSelector`'un daha esnek bir versiyonudur ve `nodeAffinity` alanında tanımlanır. İki tür kısıtlama vardır:

1. **RequiredDuringSchedulingIgnoredDuringExecution**
   - **Zorunlu (hard) bir kısıtlama** oluşturur.
   - Eğer belirttiğin düğüm kriterleri sağlanmazsa, Pod asla çalıştırılmaz.
   - `nodeSelector` gibi çalışır ama daha güçlü eşleşme kuralları sağlar.
2. **PreferredDuringSchedulingIgnoredDuringExecution**
   - **Tercihe bağlı (soft) bir kısıtlama** oluşturur.
   - Pod, belirtilen düğüme yerleşmeye çalışır ama mümkün değilse başka bir düğüme yerleşebilir.
   - Ağırlık (weight) belirterek tercih sıralaması yapabilirsin.

Örnek Kullanım

Aşağıdaki YAML dosyasında, Pod sadece **"intel" işlemciye sahip** düğümlerde çalıştırılacaktır.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: cpu-type
            operator: In
            values:
            - intel
  containers:
```

```
  - name: my-container
    image: nginx
```

Eğer `preferredDuringSchedulingIgnoredDuringExecution` kullanırsan, Pod öncelikli olarak `"intel"` işlemcili düğümlere yerleşmeye çalışır, ancak başka düğümlerde de çalışabilir.

Pod Affinity (Pod Bağımlılığı) & Pod Anti-Affinity

Pod Affinity, belirli Pod'ların **aynı düğümde veya aynı bölgedeki düğümlerde** çalışmasını sağlar.

Pod Anti-Affinity ise tam tersidir: **Belirli Pod'ların aynı düğümde olmamasını** sağlar.

Çalışma Mantığı

Pod Affinity, `podAffinity` alanında tanımlanır ve belirli etiketlere sahip Pod'ların **aynı düğümlere veya aynı topology alanlarına (örneğin, availability zone)** yerleşmesini sağlar.

Pod Anti-Affinity ise `podAntiAffinity` ile tanımlanır ve belirli etiketlere sahip Pod'ların **birbirinden uzak tutulmasını** sağlar.

Örnek Kullanımlar

1. Pod Affinity (Aynı Düğüme Yakınlaştırma)

Örneğin, web uygulaması Pod'larının birbiriyle aynı düğümde çalışmasını istiyorsan:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
```

```
        matchLabels:
          app: web
      topologyKey: "kubernetes.io/hostname"
  containers:
  - name: my-container
    image: nginx
```

Bu örnekte:

- `app: web` etiketi olan Pod'lar **aynı düğümde çalıştırılacaktır**.
- `topologyKey: "kubernetes.io/hostname"` ile **aynı düğümde olmasını** zorunlu kıldık.

2. Pod Anti-Affinity (Aynı Düğüme Gelmesini Engelleme)

Eğer **yük dengeleme (load balancing) yapmak** ve **aynı servise ait Pod'ları farklı düğümlere dağıtmak** istiyorsan:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
          matchLabels:
            app: web
        topologyKey: "kubernetes.io/hostname"
  containers:
  - name: my-container
    image: nginx
```

Bu örnekte:

- `app: web` etiketi olan Pod'lar **farklı düğümlerde çalışacaktır**.
- `topologyKey: "kubernetes.io/hostname"` ile **aynı düğümde çalışmasını engelledik**.

Taint ve Toleration (Leke ve Tolerans)

Taint ve Toleration, Kubernetes'te **belirli düğümlere (nodes) hangi Pod'ların yerleşebileceğini kontrol etmek** için kullanılır.

- **Taint (Leke):** Bir düğüme eklenir ve belirli Pod'ların o düğümde çalışmasını engeller.
- **Toleration (Tolerans):** Pod'a eklenir ve belirli taint'lere sahip düğümlerde çalışmasına izin verir.

Bu sistem sayesinde, **bazı düğümleri belirli iş yükleri için ayırabilir veya belirli Pod'ların yalnızca belirli düğümlerde çalışmasını sağlayabilirsin**.

Taint (Leke) Nedir?

Taint, bir düğüme eklenen bir **etikettir** ve o düğüme **istenmeyen Pod'ların yerleşmesini engeller**.

Bir taint, **üç bileşenden** oluşur:

1. **Key (Anahtar)** → Taint'in adı
2. **Value (Değer)** → Opsiyonel bir açıklama
3. **Effect (Etkisi)** → Taint'in davranışını belirler

Taint Kullanımı

Bir düğüme taint eklemek için aşağıdaki komutu kullanabilirsin:

```
kubectl taint nodes <node-name> key=value:Effect
```

Taint Effect (Etkileri)

Effect	Açıklama
`NoSchedule`	Pod, düğüme **kesinlikle yerleşemez**, ancak zaten oradaysa çalışmaya devam eder.
`PreferNoSchedule`	Pod, mümkünse düğüme yerleşmez ama başka seçenek yoksa yerleşebilir.

NoExecute	Pod, **bu düğümden hemen atılır** ve yerleşmesi engellenir.

Örnek: Belirli Pod'ları Engelleme

Aşağıdaki komut, **"gpu=true"** taint'ini **node01** düğümüne ekler ve **NoSchedule** etkisini kullanır:

```
kubectl taint nodes node01 gpu=true:NoSchedule
```

Bu, **sadece "gpu" kullanan ve toleration içeren Pod'ların bu düğüme yerleşmesini sağlar**.

Toleration (Tolerans) Nedir?

Toleration, bir Pod'un belirli taint'lere sahip düğümlerde çalışmasına izin verir.

Pod, **taint'i tolere ediyorsa**, Kubernetes bu düğüme Pod'u yerleştirebilir.

Toleration Kullanımı

Aşağıdaki YAML dosyası, **"gpu=true"** taint'ini tolere eden bir Pod'u tanımlar:

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  tolerations:
  - key: "gpu"
    operator: "Equal"
    value: "true"
    effect: "NoSchedule"
  containers:
  - name: my-container
    image: nginx
```

Bu Pod, **sadece "gpu=true" taint'ine sahip düğümlerde çalışabilir**.

Taint ve Toleration Çalışma Mantığı

1. **Taint, düğüme eklenir** → O düğüme rastgele Pod'ların yerleşmesini engeller.
2. **Eğer bir Pod toleration içeriyorsa**, o düğüme yerleşebilir.
3. **Eğer bir Pod toleration içermiyorsa**, Kubernetes scheduler o düğüme Pod'u yerleştirmez.

> Not: Toleration, sadece taint'in etkisini kaldırır ama Pod'un o düğüme atanmasını garanti etmez!
>
> Bunun için **Node Affinity** kullanmalısın.

Taint ve Toleration Kullanım Senaryoları

Senaryo	Çözüm
Özel donanımlı (GPU, SSD) düğümler sadece belirli iş yüklerini çalıştırmalı	**Bu düğümlere taint ekleyip, uygun Pod'lara toleration tanımla.**
Yüksek öncelikli Pod'lar belirli düğümlerde çalışmalı, diğerleri çalışmamalı	**Öncelikli düğümlere taint ekle, öncelikli Pod'lara toleration ekle.**
Bozuk bir düğüme yeni Pod'ların yerleşmesini engelleme	**Bu düğüme `NoSchedule` taint ekle.**
Bir Pod başarısız olursa, yeni Pod'ların hemen başka düğümlere gitmesini sağlama	**Bu düğüme `NoExecute` taint ekle.**

DaemonSet Ne İçin Kullanılır?

DaemonSet genellikle **her düğümde çalışması gereken sistem bileşenlerini veya agent'ları dağıtmak** için kullanılır.

Örnek Kullanım Senaryoları

Senaryo	Açıklama
Log toplama (Log Collector)	Fluentd, Filebeat gibi log toplama agent'larını her düğümde çalıştırmak.

Monitoring & Metrics	Prometheus Node Exporter, Datadog Agent gibi monitoring araçlarını dağıtmak.
Ağ Bileşenleri (Networking Daemonları)	CNI (Calico, Flannel) gibi ağ bileşenlerini her düğümde çalıştırmak.
Security & Compliance	Falco, Sysdig gibi güvenlik araçlarını her düğümde dağıtmak.

DaemonSet Nasıl Çalışır?

- Kubernetes **tüm uygun düğümlerde** otomatik olarak bir Pod çalıştırır.
- **Yeni bir düğüm cluster'a eklendiğinde**, DaemonSet bu düğümde de otomatik olarak bir Pod başlatır.
- **Bir düğüm cluster'dan çıkarılırsa**, ilgili Pod da silinir.
- **DaemonSet sadece belirli düğümlerde çalışacak şekilde de konfigüre edilebilir.**

DaemonSet Örnek Konfigürasyonu

Aşağıda, **tüm düğümlerde bir Nginx Pod'u çalıştıran** bir DaemonSet YAML dosyası var:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
```

```
    containers:
    - name: nginx
      image: nginx
```

Bu YAML dosyası:

- **Cluster'daki her düğümde** bir `nginx` Pod'u çalıştırır.
- Yeni düğümler eklendiğinde, otomatik olarak yeni düğümde bir kopya başlatır.

DaemonSet'te Belirli Düğümlerde Çalıştırma

Eğer DaemonSet'in sadece belirli düğümlerde çalışmasını istiyorsan, **nodeSelector veya nodeAffinity** kullanabilirsin.

Örnek: Sadece "worker" Etiketine Sahip Düğümlerde Çalıştırma

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        node-role.kubernetes.io/worker: "true"
      containers:
      - name: nginx
        image: nginx
```

Bu yapılandırma:

- **Sadece “worker” rolüne sahip düğümlerde** çalıştırılır.
- **Master düğümlerinde çalışmaz.**

Eğer daha esnek kurallar istiyorsan, **Node Affinity** kullanabilirsin.

DaemonSet’i Yönetmek İçin Komutlar

DaemonSet’leri yönetmek için aşağıdaki `kubectl` komutlarını kullanabilirsin:

DaemonSet Listeleme

```
kubectl get daemonsets
```

Belirli Bir DaemonSet Detaylarını Görme

```
kubectl describe daemonset <daemonset-ismi>
```

DaemonSet Silme

```
kubectl delete daemonset <daemonset-ismi>
```

DaemonSet vs Deployment Farkı

Özellik	DaemonSet	Deployment
Replika Sayısı	Her düğümde 1 Pod	Belirtilen sayıda Pod
Yeni Düğümlerde Otomatik Çalışma	Evet	Hayır
Genellikle Kullanım Amacı	Sistem bileşenleri, monitoring, log toplama	Uygulama servisleri
Özel Düğümlerde Çalıştırma	nodeSelector, Affinity ile mümkün	Evet, ama manuel ayarlamak gerekir

Persistent Volume (PV)

- Küme yöneticisi tarafından sağlanan **fiziksel depolama alanıdır**.
- Bir **StorageClass** ile ilişkilendirilebilir veya doğrudan statik olarak tanımlanabilir.
- Fiziksel disk, NFS, AWS EBS, Azure Disk, Google Persistent Disk gibi **dış depolama kaynaklarını** kullanabilir.
- Bağımsız bir kaynak olup, belirli bir **pod'a bağlı değildir**.
- Hayatta kalma politikaları (**Reclaim Policy**) olabilir:
  - `Retain`: PV silinse bile verileri saklar.
  - `Delete`: PV silindiğinde verileri de siler.
  - `Recycle`: Eski verileri temizleyip tekrar kullanılabilir hale getirir.

Persistent Volume Claim (PVC)

- **Pod'ların PV talep etmesini** sağlayan nesnedir.
- Belirli bir depolama kapasitesi ve erişim modu talep eder:
  - `ReadWriteOnce (RWO)`: Tek bir node yazabilir.
  - `ReadOnlyMany (ROX)`: Birden çok node yalnızca okuyabilir.
  - `ReadWriteMany (RWX)`: Birden çok node yazabilir.
- **PVC, uygun bir PV bulursa bağlanır**, eğer uygun PV yoksa yeni bir PV oluşturulmasını talep edebilir (Dynamic Provisioning).

Örnek YAML Dosyaları

1. Persistent Volume (PV) Tanımı:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
```

```
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/mnt/data"
```

Bu, `5Gi` büyüklüğünde, `ReadWriteOnce` erişim moduna sahip ve **kümeye bağlı bir hostPath dizininde bulunan** bir PV tanımlar.

2. Persistent Volume Claim (PVC) Tanımı:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Bu PVC, en az `5Gi` depolama alanı ve `ReadWriteOnce` erişim modu talep eder. Eğer kümede uygun bir PV varsa otomatik olarak bağlanır.

3. PVC Kullanarak Pod Tanımlama:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
    - name: example-container
```

```
    image: nginx
    volumeMounts:
      - mountPath: "/usr/share/nginx/html"
        name: example-storage
  volumes:
    - name: example-storage
      persistentVolumeClaim:
        claimName: example-pvc
```

Bu pod, `example-pvc` üzerinden gelen depolamayı `/usr/share/nginx/html` yoluna monte eder.

StorageClass Nedir?

StorageClass, Kubernetes'te **dinamik (dynamic) depolama sağlamak** için kullanılan bir nesnedir. Yani, **Persistent Volume (PV)** oluşturmayı otomatikleştiren bir mekanizmadır.

Normalde, **Persistent Volume (PV)** nesnelerini **statik** olarak tanımlamak gerekir. Ancak, **StorageClass** kullanarak, **Persistent Volume Claim (PVC)** oluşturulduğunda **otomatik olarak bir PV oluşturulmasını sağlayabilirsin**.

StorageClass'in Temel Özellikleri

1. **Dinamik PV sağlama** (Dynamic Provisioning) → Kullanıcı PVC oluşturduğunda uygun bir PV otomatik olarak oluşturulur.
2. **Farklı depolama sistemleri desteklenir** (AWS EBS, Azure Disk, Google Persistent Disk, NFS, Ceph, vs.).
3. **Farklı depolama politikaları (reclaim policy) belirlenebilir**.
4. **Performans ve erişim özellikleri tanımlanabilir**.

StorageClass YAML Örneği

Aşağıda bir **StorageClass** tanımı bulunuyor. Bu, Kubernetes'in **dinamik olarak PV oluşturmasına** olanak tanır.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-storage
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
```

Bu YAML dosyasında neler var?

- `provisioner`: Kubernetes'in hangi depolama sağlayıcısını (provisioner) kullanacağını belirtir. (Örn: `kubernetes.io/aws-ebs`, `kubernetes.io/gce-pd`, `kubernetes.io/csi`)
- `parameters`: Depolama sınıfının parametrelerini belirtir (örneğin, AWS'de `gp2` tipi disk kullanılacak).
- `reclaimPolicy`: PV silindiğinde ne olacağını belirler:
  - `Retain` → PV silinse bile veri korunur.
  - `Delete` → PVC silindiğinde PV de otomatik olarak silinir.
- `allowVolumeExpansion`: `true` olarak ayarlandığında, PVC genişletilebilir.
- `volumeBindingMode`:
  - `Immediate`: PV, PVC talep edildiği anda bağlanır.
  - `WaitForFirstConsumer`: Pod'un talep etmesini bekler (Node ve bölgeyi doğru seçmek için).

StorageClass ile PVC Kullanımı

Eğer kümede **StorageClass** tanımlıysa, aşağıdaki gibi bir **PVC tanımlayabilirsin**:

```
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: fast-storage
```

- `storageClassName: fast-storage` → **fast-storage** adındaki StorageClass kullanılacak.
- PVC oluşturulduğunda Kubernetes **otomatik olarak bir PV oluşturacak** ve bunu PVC'ye bağlayacak.

StatefulSet Nedir?

StatefulSet, Kubernetes'te **durum bilgisi olan (stateful) uygulamaları yönetmek için kullanılan** bir iş yükü (workload) nesnesidir. **Deployment ve ReplicaSet gibi pod yönetimi sağlar, ancak ek olarak pod'ların kimliklerini ve verilerini korur.**

StatefulSet'in Özellikleri

1. **Sabit Bir Kimlik (Stable Identity)**
   - **Her pod'un kendine özgü bir ismi ve sırası vardır.**
   - Pod'lar `<StatefulSet adı>-<index>` formatında adlandırılır.
   - Örneğin: `web-0`, `web-1`, `web-2` gibi.
2. **Sabit Bir Depolama (Stable Storage)**
   - **Her pod'un kendine özel bir Persistent Volume (PV) vardır.**
   - PVC, **pod silinse bile verileri kaybetmez**.
   - **Pod yeniden başlasa bile aynı veriyi kullanır.**
3. **Sıralı Başlatma ve Kapatma (Ordered Startup & Shutdown)**

- **Pod'lar sırasıyla başlatılır ve sırasıyla silinir.**
- Örneğin: `web-0` başlamadan `web-1` başlamaz.

4. **DNS Üzerinden Erişim (Stable Network Identity)**
   - **Pod'lar için benzersiz DNS kayıtları atanır.**
   - Örneğin, `web-0.my-service.default.svc.cluster.local`.
5. **Dağıtılmış Sistemler İçin İdealdir**
   - **Kafka, Zookeeper, MongoDB, MySQL gibi durum bilgisi gerektiren uygulamalar için kullanılır.**
   - Çünkü **her pod'un sırası, kimliği ve depolama alanı önemlidir.**

StatefulSet Kullanım Senaryoları

- **Veritabanları**: PostgreSQL, MySQL, MongoDB
- **Dağıtılmış sistemler**: Kafka, Zookeeper, RabbitMQ
- **Cache sistemleri**: Redis, etcd
- **Ağ tabanlı uygulamalar**: IP kimliği gerektiren uygulamalar

---

StatefulSet YAML Örneği

Aşağıdaki örnek, **3 replikalı bir StatefulSet oluşturur.**

1️⃣ Persistent Volume (PV) için StorageClass Tanımla

Eğer **dinamik depolama** kullanacaksan, önce bir **StorageClass** oluşturman gerekir:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: my-storage
provisioner: kubernetes.io/aws-ebs
parameters:
```

```
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
```

2️⃣ StatefulSet Tanımı

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "web"
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: web
        image: nginx
        ports:
        - containerPort: 80
          name: http
        volumeMounts:
        - name: www
          mountPath: "/usr/share/nginx/html"
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
```

```
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
    storageClassName: my-storage
```

Bu YAML Ne Yapıyor?

- **3 adet `web` pod'u başlatıyor** (`web-0`, `web-1`, `web-2`).
- **Her pod için ayrı bir Persistent Volume (PV) oluşturuyor**.
- **Pod'ların her biri `nginx` çalıştırıyor** ve `/usr/share/nginx/html` dizinine veri yazabiliyor.
- **Pod'ların DNS adresleri şöyle olacak:**
  - `web-0.web.default.svc.cluster.local`
  - `web-1.web.default.svc.cluster.local`
  - `web-2.web.default.svc.cluster.local`

StatefulSet vs. Deployment

Özellik	Deployment	StatefulSet
Pod İsimleri	Rastgele oluşturulur	Sabit (`web-0`, `web-1`, vb.)
Depolama	Pod silinirse veri kaybolur	Her pod'un sabit bir PV'si var
Başlatma ve Silme Sırası	Aynı anda olur	Sıralı başlatma/kapatma
Ağ Kimliği (DNS)	Rastgele atanır	Her pod'un sabit bir DNS adresi var

Job Nedir?

Kubernetes'te **Job**, **belirli bir işi tamamlamak için** çalışan pod'ları yöneten bir nesnedir.

Job nesnesi, **geçici (short-lived) işler için** kullanılır ve **tamamlandığında sona erer**.

Önemli Özellikler:

✅ **Tek seferlik veya belirli sayıda çalıştırılacak işleri yönetir.**

✅ **Başarıyla tamamlanan pod'ları tekrar başlatmaz.**

✅ **Pod başarısız olursa, yeniden başlatıp görevi tamamlamaya çalışır.**

✅ **Batch processing (toplu işlem) ve arka plan işlemleri için idealdir.**

Job Kullanım Senaryoları

- **Veri İşleme**: Büyük veri kümeleri üzerinde analiz yapmak.
- **Yedekleme İşlemleri**: Veritabanı yedekleme işleri.
- **Tek Seferlik Script Çalıştırma**: Migration, batch işlemleri.
- **Makine Öğrenmesi Model Eğitimi**: Model eğitme işlemlerini tamamladıktan sonra pod'un kapanması.

Job YAML Örneği

Aşağıdaki Job, bir pod başlatır ve pod, `echo "Job tamamlandı"` komutunu çalıştırıp bittiğinde sona erer.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: example-job
spec:
  template:
    metadata:
      name: example-pod
    spec:
      restartPolicy: Never
      containers:
      - name: example-container
        image: busybox
        command: ["echo", "Job tamamlandı"]
```

📌 **Bu Job, tek bir pod başlatır ve pod işlemi tamamlandığında kapanır.**

📌 `restartPolicy: Never` sayesinde pod başarısız olursa Kubernetes yeni bir pod başlatır.

Paralel Job (Multiple Pods)

Job nesnesini **birden fazla pod paralel çalıştıracak şekilde** yapılandırabilirsin.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: parallel-job
spec:
  completions: 5  # Toplam 5 pod çalıştıracak
  parallelism: 2  # Aynı anda en fazla 2 pod çalıştıracak
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: example-container
        image: busybox
        command: ["echo", "Parallel Job çalışıyor"]
```

📌 `completions: 5` → Toplamda 5 pod çalıştırılacak.

📌 `parallelism: 2` → Aynı anda en fazla 2 pod çalışacak.

📌 Kubernetes **Job'in tamamlanması için 5 pod'un da bitmesini bekleyecek**.

CronJob Nedir?

Eğer **Job'in belirli zaman aralıklarında çalışmasını** istiyorsan, **CronJob** kullanabilirsin.

Örnek: **Her gün saat 12:00'de çalışacak bir Job**:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: daily-job
spec:
  schedule: "0 12 * * *"  # Her gün öğlen 12:00'de çalışır
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
          - name: example-container
            image: busybox
            command: ["echo", "Her gün çalışan CronJob"]
```

📌 **CronJob, Job'in belirli periyotlarla çalışmasını sağlar.**

📌 **Linux'taki `cron` sistemi gibi zamanlanmış görevler çalıştırır.**

Job vs Deployment vs DaemonSet

Özellik	Job	Deployment	DaemonSet
Çalışma Süresi	Geçici, tamamlanınca durur	Sürekli çalışır	Her node'da sürekli çalışır
Ölçeklenme	Tek seferlik veya belirli sayıda çalıştırılır	Pod'ları ölçekleyerek artırabiliriz	Node sayısına bağlı
Pod Sayısı	Belirli sayıda çalıştırılabilir (`completions`)	Kullanıcı belirler (`replicas: X`)	Node sayısına bağlı
Tekrar Çalıştırma	Başarısız olan pod'u tekrar başlatır	Pod ölürse yeni pod başlatır	Yeni node eklenirse yeni pod çalıştırır
Kullanım Alanı	Batch işlemleri, veri işleme, tek seferlik işler	Web uygulamaları, API'ler	Log toplama, monitoring, ağ bileşenleri

Kubernetes Authentication (Kimlik Doğrulama)

Kubernetes'te kimlik doğrulama, kullanıcıların ve servislerin Kubernetes API'ye erişmeden önce kimliklerini doğrulaması işlemidir.

Adım Adım Authentication Süreci:

1. Öncelikle bir Private Key ve CSR oluşturulur:

```
openssl genrsa -out mustafasenlik.key 2048

openssl req -new -key mustafasenlik.key -out mustafasenlik.csr -subj "/CN=
mustafasenlik/O=DevTeam"
```

Buradaki:

- `CN`: Kullanıcı adını temsil eder.
- `O`: Grubu (organizasyonu) temsil eder.

---

2. CertificateSigningRequest Kubernetes'e gönderilir:

CSR Kubernetes'e gönderilip onay istenir.

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: mustafasenlik
spec:
  groups:
  - system:authenticated
  request: $(cat mustafasenlik.csr | base64 | tr -d "\n")
  signerName: kubernetes.io/kube-apiserver-client
  usages:
  - client auth
EOF
```

---

3. CSR onaylanır ve sertifika (CRT) alınır:

```
kubectl certificate approve mustafasenlik

kubectl get csr mustafasenlik -o jsonpath='{.status.certificate}' | base64 -d >
> mustafasenlik.crt
```

Artık Kubernetes API, kullanıcı kimliğini tanıyabilir.

---

4. kubectl config ayarları yapılır:

```
kubectl config set-credentials mustafasenlik --client-certificate=mustafasenli
k.crt --client-key=mustafasenlik.key

kubectl config set-context mustafasenlik-context --cluster=minikube --user=
mustafasenlik

kubectl config use-context mustafasenlik-context
```

Artık Kubernetes'e bağlanırken kimliğin doğrulanır.

RBAC Nedir?

RBAC (Role-Based Access Control), Kubernetes içinde kullanıcı veya servis hesaplarının (ServiceAccounts) hangi kaynaklara, hangi yetkilerle erişebileceğini belirleyen yetkilendirme yöntemidir.

Kubernetes RBAC Temel Kavramları:

1. Role

Belirli bir namespace içindeki izinleri tanımlar.

Örnek bir Role:

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

> Bu Role sadece dev namespace içindeki pod'ları okumaya izin verir.

---

2. ClusterRole

Namespace bağımsız, tüm cluster üzerinde geçerli izinleri tanımlar.

Örnek bir ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: node-reader
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "watch", "list"]
```

> Bu ClusterRole, cluster genelindeki tüm node'ları okuma yetkisi sağlar.

---

3. RoleBinding

Bir **Role**'ü kullanıcıya veya servis hesabına bağlayarak izinleri devreye sokar.

(Sadece ilgili namespace içinde etkilidir.)

Örnek RoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: dev
subjects:
- kind: User
  name: mustafasenlik
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

> Bu RoleBinding ile mustafasenlik kullanıcısı, dev namespace'inde podları okuyabilir.

---

4. ClusterRoleBinding

Bir **ClusterRole**'ü kullanıcıya veya servis hesabına bağlayarak izinleri tüm cluster'da devreye sokar.

Örnek ClusterRoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: node-reader-binding
subjects:
- kind: User
  name: mustafasenlik
```

```
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: node-reader
  apiGroup: rbac.authorization.k8s.io
```

> Bu ClusterRoleBinding sayesinde mustafasenlik, tüm cluster üzerindeki node'ları okuyabilir.

Kubernetes ServiceAccount Nedir?

ServiceAccount, Kubernetes üzerinde çalışan pod'ların, Kubernetes API'ye güvenli şekilde erişmesini sağlayan, **makineler (pod'lar)** için tasarlanmış bir kullanıcı hesabıdır.

ServiceAccount Ne İşe Yarar?

- **Pod'ların Kubernetes API ile etkileşime geçmesini sağlar.**
- Pod'ların kimlik doğrulaması yapmasını sağlar.
- Pod'lar için ayrı yetkilendirme (RBAC) politikaları oluşturulmasına izin verir.

ServiceAccount Nasıl Oluşturulur?

Örnek YAML:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: mustafasenlik-sa
  namespace: dev
```

```
kubectl apply -f serviceaccount.yaml
```

ServiceAccount'u Pod ile Kullanma

Bir pod tanımında şu şekilde belirtilir:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
  namespace: dev
spec:
  serviceAccountName: mustafasenlik-sa
  containers:
  - name: demo
    image: nginx
```

> Bu pod artık belirtilen ServiceAccount ile Kubernetes API'ye erişir.

ServiceAccount ve RBAC

Bir ServiceAccount’a RBAC ile yetki atamak için `RoleBinding` ya da `ClusterRoleBinding` kullanılır.

Örnek RoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: dev
subjects:
- kind: ServiceAccount
```

```
  name: mustafasenlik-sa
  namespace: dev
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

> Bu ServiceAccount artık pod'ları listeleyip izleyebilir.

Kubernetes Ingress Controller Nedir?

Ingress Controller, Kubernetes üzerinde çalışan uygulamalara **cluster dışından gelen HTTP/HTTPS trafiğini** yönetip yönlendiren bir bileşendir.

Ne İşe Yarar?

- Birden fazla servise tek noktadan dışarıya erişim sağlar.
- Alan adına (host) veya URL yoluna göre trafiği ilgili servise yönlendirir.
- SSL/TLS sertifikalarını yönetir ve HTTPS bağlantıları kurar.
- Yük dengeleme (load balancing) yapar.

Nasıl Çalışır?

Şöyle özetleyebiliriz:

```
Kullanıcı → Ingress Controller → Ingress kurallarına göre ilgili Service → Pod
```

- **Ingress Controller** gelen isteği kabul eder.
- Kubernetes içinde tanımlanmış **Ingress** kurallarını kontrol eder.
- Gelen URL ya da alan adına göre isteği doğru servise iletir.

Örnek Bir Ingress Tanımı:

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

> Bu örnekte mustafa.com alan adına gelen istekler, demo-service adlı servise yönlendirilir.

Yaygın Ingress Controller'lar:

- **NGINX Ingress Controller** (en popüler)
- **Traefik**
- **HAProxy**
- **Istio (Service Mesh)**

k8s Dashboard ve GUI

1. Minikube Başlat

```
minikube start
```

Eğer özel bir driver (örneğin `docker`) kullanıyorsan:

```
minikube start --driver=docker
```

---

2. Metrics Server'ı Yükle

Minikube için uyumlu olan sürüm:

```
minikube addons enable metrics-server
```

Bu komut, Minikube'un içine otomatik olarak Metrics Server'ı kurar.

---

3. Dashboard (GUI) Başlat

```
minikube dashboard
```

Bu komut hem Dashboard'u başlatır hem de varsayılan tarayıcında otomatik olarak açar.

---

4. Tarayıcı Açılmıyorsa

Eğer otomatik açılmazsa, terminalde verilen URL'yi kopyalayıp tarayıcıya yapıştırabilirsin.

Örn:

```
http://127.0.0.1:XXXXX/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/
```

Lens Nedir?

Lens, Kubernetes yönetimi için kullanılan açık kaynaklı bir **GUI (grafik arayüz)** aracıdır.

`kubectl` komutları yerine görsel bir ortamda cluster yönetimi yapmanı sağlar.

Neden Tercih Edilir?

- **Ekstra GUI aracı**: Minikube Dashboard dışında daha güçlü bir alternatif.
- **Multi-cluster desteği**: Birden fazla küme aynı anda yönetilebilir.
- **Kaynak izleme**: Pod, node, namespace, CPU/RAM kullanımı vs. görsel olarak izlenir.
- **YAML düzenleme ve log izleme**: GUI üzerinden kolay düzenleme ve hata takibi yapılır.
- **Gelişmiş kullanıcılar için**: RBAC, Prometheus entegrasyonu gibi gelişmiş özellikler sunar.

imagePullPolicy nedir?

`imagePullPolicy`, Kubernetes'te bir Pod oluşturulurken container image'ının ne zaman çekileceğini belirler.

3 farklı seçenek vardır:

1. **Always**

   Her pod çalıştırıldığında image'ı registry'den tekrar çeker.

   En güncel image'ı almak istiyorsan bu kullanılır.

   Eğer image tag'in `:latest` ise Kubernetes otomatik olarak bunu seçer.

2. **IfNotPresent**

   Node üzerinde image zaten varsa tekrar çekmez.

   Sadece image yoksa çeker.

   Geliştirme ortamlarında tercih edilir.

3. **Never**

   Hiçbir durumda image'ı registry'den çekmez.

Image node'a daha önceden manuel olarak yüklenmiş olmalı.

Offline ortamlarda kullanılır.

Kubernetes varsayılan davranışı:

- Image tag `:latest` ise policy `Always` olur.
- Diğer tag'lerde `IfNotPresent` kullanılır.

---

imagePullSecrets nedir?

Özel bir container registry'den image çekebilmek için Kubernetes'e kimlik bilgisi vermen gerekir.

Bu kimlik bilgileri `imagePullSecrets` aracılığıyla tanımlanır.

1. Önce bir secret oluşturursun:

```
kubectl create secret docker-registry regcred \
  --docker-server=https://index.docker.io/v1/ \
  --docker-username=kullaniciadi \
  --docker-password=sifre \
  --docker-email=email@example.com
```

1. Bu secret'ı pod tanımında belirtirsin:

```
spec:
  containers:
  - name: myapp
    image: myregistry.com/myapp:v1
  imagePullSecrets:
  - name: regcred
```

Bu şekilde Kubernetes özel registry'e giriş yaparak image'ı çeker.

Static Pod nedir?

Static Pod, Kubernetes'te doğrudan kubelet tarafından yönetilen bir pod türüdür. Normal pod'lar kube-apiserver üzerinden çalıştırılırken, static pod'lar kubelet'in kendisi tarafından başlatılır ve izlenir. Yani cluster'a değil, doğrudan node'a bağlıdır.

Nerede kullanılır?

- Kubernetes kontrol bileşenleri (kube-apiserver, scheduler, controller-manager) static pod olarak çalışır.
- Yüksek güvenilirlik ve doğrudan node başlatımı gereken yerlerde tercih edilir.
- Cluster kurulumu ve bootstrap aşamasında kullanılır.

Nasıl çalışır?

1. Static pod YAML dosyası belirli bir dizine yerleştirilir. Bu dizin genelde:

   `/etc/kubernetes/manifests`
2. Kubelet bu dizini sürekli izler. Dosya eklenirse pod başlatılır, silinirse pod durdurulur.
3. Bu pod'lar `kubectl get pods` ile görünebilir, ama kontrol kube-apiserver'da değildir.

Özellikleri:

- Replication veya autoscaling yoktur.
- Sadece o node üzerinde çalışır.
- `kubectl delete` ile silsen bile kubelet tekrar başlatır.
- Tamamen YAML dosyasının varlığına bağlıdır.

Örnek static pod dosyası:

```
apiVersion: v1
kind: Pod
metadata:
  name: static-nginx
spec:
  containers:
```

```
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Bu dosyayı `/etc/kubernetes/manifests/` içine koyduğunda, kubelet pod'u başlatır.

NetworkPolicy nedir?

Kubernetes'te pod'lar varsayılan olarak birbirine sınırsız erişebilir.

NetworkPolicy, bu iletişimi sınırlamak için kullanılan güvenlik kuralıdır.

Yani:

"Şu pod şuna erişsin ama öbürüne erişmesin" gibi kurallar yazmamızı sağlar.

Ne işe yarar?

- Pod'lar arası trafiği kontrol eder.
- Belirli IP'lerden veya namespace'lerden gelen bağlantıları sınırlar.
- Hem **gelen (ingress)** hem **giden (egress)** trafiği yönetebilir.
- Mikroservis mimarilerinde güvenlik katmanı oluşturur.
- İstemediğin bir pod'un, hassas bir servise erişmesini engeller.

Nasıl çalışır?

1. Belirli bir pod grubunu hedef alırsın. (`podSelector`)
2. Bu pod grubuna kimlerin erişebileceğini tanımlarsın. (`from`)
3. Dilersen bu pod grubunun dışarıya erişimini de kısıtlayabilirsin. (`to`)
4. Sadece açıkça izin verilen trafik geçebilir, geri kalan **engellenir**.

Çalışması için ne gerekir?

- Kullandığın Kubernetes ağ eklentisinin (CNI) **NetworkPolicy desteklemesi gerekir.**
- Flannel desteklemez. Calico, Cilium, Weave gibi plugin'ler destekler.

Örnek: Sadece frontend, backend'e erişebilsin

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: backend
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
```

Açıklaması:

- `role=backend` olan pod'lara sadece `role=frontend` olan pod'lar erişebilir.
- Diğer pod'lar, bu backend'e erişemez.

Varsayılan davranış:

- Hiç NetworkPolicy yoksa → herkes herkese erişebilir.
- Bir pod için en az 1 NetworkPolicy tanımlandıysa → sadece izin verilen trafik geçer, kalan **engellenir**.

Helm ne işe yarar?

📎 https://helm.sh/

Kubernetes manifest dosyaları (yaml'lar) genelde çok parçalı ve karmaşıktır.

Helm bu manifest dosyalarını **chart** adı verilen paketler haline getirir.

Sen de bu paketleri indirip, kurup, güncelleyip, kaldırabilirsin. Tıpkı bir yazılım paketi gibi.

Helm ile ne yapabilirsin?

- Hazır uygulamaları (nginx, prometheus, mysql, vs.) tek komutla kurarsın.
- Değişkenlerle esnek yapı kurarsın (`values.yaml` dosyası sayesinde).
- Karmaşık uygulamaları daha kolay yönetirsin.
- Versiyon kontrolüyle geçmişe dönebilirsin.
- CI/CD süreçlerine rahat entegre edersin.

Örnek kullanım:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install my-nginx bitnami/nginx
```

Bu komutla bitnami reposundan nginx kurulumu yapılır.

Helm'in yapısı:

1. **Chart**: Bir uygulamayı tanımlayan dosya grubu
2. **Release**: Chart'ın cluster üzerinde çalışır hali
3. **Repository**: Chart'ların tutulduğu yer (örn. Bitnami, ArtifactHub)
4. **values.yaml**: Değiştirilebilir ayarların bulunduğu dosya

---

Ne zaman kullanılır?

- Aynı uygulamayı farklı ortamlara kurarken (dev, test, prod)
- Uygulamanın farklı konfigürasyonlara ihtiyaç duyduğu senaryolarda
- Karmaşık servisleri kolayca dağıtmak istediğinde

- GitOps ve CI/CD sistemlerinde otomasyon için

Prometheus Stack nedir?

Prometheus Stack, şu bileşenlerden oluşan bir gözlemleme sistemidir:

1. **Prometheus** → Metrik verileri toplar ve saklar
2. **Grafana** → Bu metrikleri görsel grafiklerle sunar
3. **Alertmanager** → Belirli koşullarda alarm üretir (mail, slack, vs.)
4. **Node Exporter / Kube State Metrics** → Kubernetes ve node verilerini Prometheus'a taşır

---

Ne işe yarar?

- Pod'ların, node'ların, service'lerin CPU, RAM, disk, ağ kullanımlarını izlersin
- Cluster içi uygulamaların durumlarını takip edersin
- Alarmlar kurup hata olduğunda bildirim alırsın
- Grafiklerle geçmiş performansı analiz edersin

Nasıl kurulur?

En yaygın yöntem: **Helm Chart** ile kurmak.

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update

helm install kube-prometheus-stack prometheus-community/kube-prometheus-stack
```

Bu komut, içinde Prometheus, Grafana, Alertmanager ve tüm yardımcı servislerin olduğu tam paketi kurar.

Sonra ne olur?

- Prometheus kendi arayüzünden sorgularla veri çeker: `http://<svc-ip>:9090`
- Grafana arayüzü gelir: `http://<svc-ip>:3000`

  (Default kullanıcı: admin / prom-operator'dan alınabilir)
- Hazır dashboard'lar üzerinden node, pod, container metriklerini anlık olarak izlersin

---

Neden kullanılır?

- Üretim ortamında **güvenilir gözlemleme** yapman şarttır
- Uygulama problemleri daha oluşmadan tespit edilir
- Gerçek zamanlı + geçmiş verilerle analiz yapılabilir
- Otomasyon ve alarm sistemleri kurabilirsin

1. EFK Stack Nedir?

EFK, Kubernetes ortamında logları merkezi bir yerde toplamak ve incelemek için kullanılan üçlü bileşendir:

- **Elasticsearch**: Log verilerini indeksler ve saklar
- **Fluentd**: Pod ve node loglarını toplar, Elasticsearch'e gönderir
- **Kibana**: Elasticsearch'teki logları görsel arayüzle sunar

Bu sistem sayesinde, pod silinse bile loglara erişim sağlanabilir.

---

2. Kurulum Ön Koşulları

- Helm yüklü olmalı
- Kubernetes cluster çalışıyor olmalı (Minikube, Kind, vs.)
- Cluster'da en az 4 GB RAM olmalı (EFK kaynak tüketimi yüksektir)

---

3. Helm Repo Ekleme

İlk olarak Elastic Helm chart reposu eklenir:

```
helm repo add elastic https://helm.elastic.co
helm repo update
```

4. Elasticsearch Kurulumu

```
helm install elasticsearch elastic/elasticsearch \
  --set replicas=1 \
  --set minimumMasterNodes=1 \
  --namespace logging --create-namespace
```

Bu komutla tek nodlu Elasticsearch kurulmuş olur. Namespace olarak `logging` kullanıldı.

5. Kibana Kurulumu

```
helm install kibana elastic/kibana \
  --set service.type=NodePort \
  --namespace logging
```

NodePort ile Kibana tarayıcıdan erişilebilir hale gelir.

Port numarasını öğrenmek için:

```
kubectl get svc -n logging kibana
```

6. Fluentd Kurulumu (Bitnami üzerinden)

Fluentd log toplamak için kullanılır. Bitnami chart'ı tercih edilebilir:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update

helm install fluentd bitnami/fluentd \
  --set aggregator.enabled=false \
```

```
  --set forwarder.enabled=true \
  --set forwarder.configMap=config \
  --namespace logging
```

ConfigMap ile log formatı ayarlanabilir. Elasticsearch adresi, Fluentd içinde tanımlanır.

---

7. Kibana'ya Erişim

Tarayıcıdan Kibana arayüzüne eriş:

```
minikube service kibana -n logging
```

Kibana ilk açıldığında bir index pattern tanımlanmalı:

Örneğin: `logstash-*`

---

8. Test Etme

Cluster içinde basit bir pod oluştur ve log üret:

```
kubectl run logger --image=busybox -it --restart=Never -- sh
```

İçerideyken sürekli log yaz:

```
while true; do echo "log test: $(date)"; sleep 1; done
```

Kibana arayüzünden bu logları filtreleyerek görebilirsin.

1. Service Mesh Nedir?

Service Mesh, Kubernetes içindeki servisler arası iletişimi yöneten bir altyapıdır.

Temel amacı, uygulama koduna dokunmadan ağ trafiğini kontrol edebilmektir.

Kullanım amaçları:

- Servisler arası TLS şifreleme

- Trafik yönlendirme ve A/B testi
- Gözlemleme ve loglama
- Yetkilendirme ve kimlik doğrulama
- Retry, timeout, circuit breaker gibi özellikler

En popüler çözüm: **Istio**

---

2. Istio Bileşenleri

- **Envoy Proxy**: Her pod'un yanına eklenen sidecar proxy
- **Istiod**: Istio'nun kontrol düzlemi (config, TLS, routing, vs.)
- **Pilot, Mixer, Citadel** gibi eski bileşenler artık Istiod içinde birleşmiştir
- **Ingress Gateway**: Dış dünyadan içeri trafik almak için kullanılan bileşen

---

3. Kurulum Ön Koşulları

- Helm yüklü olmalı
- Kubernetes cluster çalışıyor olmalı
- 4 GB+ RAM önerilir
- `istioctl` CLI aracı indirilmeli

İndirmek için:

```
curl -L https://istio.io/downloadIstio | sh -
cd istio-*/bin
export PATH=$PWD:$PATH
```

---

4. Istio CRD'leri ve Base Bileşenlerini Yükleme

```
istioctl install --set profile=demo -y
```

Bu komut, Istio'yu "demo" profiliyle hızlıca kurar. Eğitim ve test amaçlı kullanılır.

---

5. Istio Etiketleme

Namespace içine Istio sidecar’larının otomatik eklenmesi için:

```
kubectl label namespace default istio-injection=enabled
```

Artık default namespace'teki tüm pod'lara otomatik Envoy sidecar eklenecek.

---

6. Uygulama Dağıtımı (Örnek Bookinfo)

```
kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml
```

Bu komut örnek bir mikroservis uygulaması (Bookinfo) kurar.

İlgili servisleri görmek için:

```
kubectl get services
kubectl get pods
```

---

7. Ingress Gateway Ayarları

İstio dışarıdan gelen trafiği yönlendirmek için Ingress Gateway kullanır.

Gateway ve VirtualService tanımla:

```
kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml
```

IP ve port bilgisi:

```
kubectl get svc istio-ingressgateway -n istio-system
```

---

8. İzleme ve Gözlemleme

Istio kurulduğunda şu araçlarla entegre gelir:

- Prometheus: metrik toplama
- Grafana: metrik görselleştirme

- Kiali: Service graph ve trafik analizi
- Jaeger: dağıtık izleme (tracing)

Erişim için:

```
istioctl dashboard kiali
istioctl dashboard grafana
istioctl dashboard jaeger
```

---

9. Temel Senaryo

Canary deployment yapmak istiyorsun. Trafiğin yüzde 90'ı eski sürüme, yüzde 10'u yeni sürüme gitsin.

Bunu sadece Istio'nun VirtualService ve DestinationRule kaynaklarıyla yapabilirsin. Uygulama koduna dokunmazsın.

---

10. Ne Zaman Kullanılır?

- Mikroservis sayısı arttığında
- Ağ trafiği karmaşık hale geldiğinde
- TLS, kimlik doğrulama, trafik yönlendirme gibi işler kod dışında çözülsün istendiğinde
- Gözlemleme ve güvenlik katmanı gerekliyse