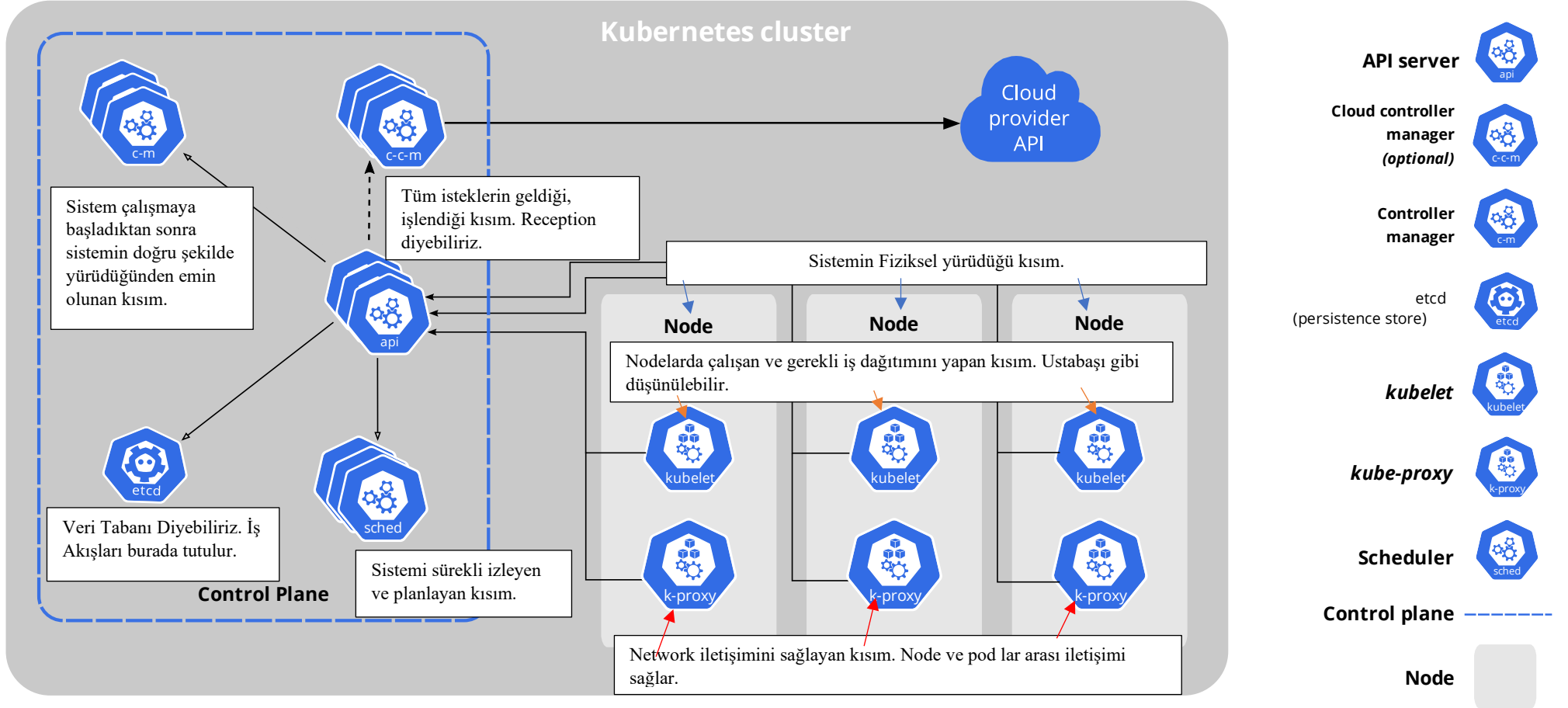


KUBERNATE

Kubernetes Architecture

Declarative Yöntem ile çalışan, Container Orchestration bir araçtır. Ne yapmak istediğinizi söylüyorsunuz o sizin adınıza tüm işleri hallediyor.



- Kube-apiserver:

Kubernetes API nı ortaya çıkartan Control Plane yapısının en önemli bileşenidir. Tüm sisteme giriş ve çıkışlar buradan yapılır. Ayrıca diğer tüm komponent ve node bileşenlerinin direkt erişimde bulunduğu tek yerdir.

Tüm istekler buradan gerçekleşirken, Authentication ve Authorization işlemlerinin yapıldığı kısımdır. İstekleri Rest Api isteği olarak alabileceği gibi “kubect!” gibi araçlarla da çalışabilir.

- Etcd:

Tüm sistem verisini tutar. MetaData verileri ve Kubernetes içinde oluşturulan tüm obje bilgilerinin tutulduğu Key-Value veri deposudur. Cluster in mevcut yapılandırmasının tamamı buradadır. Kube-ApiServer dışında diğer bileşenler etcd ile haberleşemezler.

- Kube-scheduler:

Yeni oluşturulan ya da bir node a ataması yapılmamış Pod ları izler ve üzerinde çalışacakları bir nodes seçen kısımdır. Aslında sistemde var olan zorunluluklar, tainler v.s. bilgilerine göre bir pod un çalışabileceği en uygun node u seçerek oraya atamasını yapar.

- Kube-controller-manager:

Her kontroller ayrı bir süreç izler, ancak karmaşıklığı azaltmak adına hepsi bir binary olarak derlenmişlerdir ve tek bir Process olarak çalışırlar. Etcd de saklanan verileri inceleyerek sistemi gözlemler, eğer istenilen durum ile hali hazırda bulunan durum arasında fark var ise sistemi istenilen ayarlarına getirir. Güncelleme, silme işlemlerini yapar. EWS, Azure gibi sistemlerin entegrasyonlarında “Cloude controler Manager” adı verilen ve yönetimi sağlayan ayrıca bir ara katman vardır. Bunları örneklemek gerekirse:

- Node controller
- Service Account & Token controller
- Endpoints controller
- Job controller

NOT: Kurbarnates yapısının yönetim kısmını oluşturan (resimdeki sol taraf) Master Node olarak isimlendirilir. Sistem yükünün ve posların çalıştığı (resimdeki sağ kısım) Worker Node olarak adlandırılır.

- Kubelet:

Cluster da her node için çalışan agent türü bir yapıdır. Pod üzerinde çalıştırılmak istenilen container(image) ların çalıştırılmasını sağlar. Kubelet, çeşitli sistemler aracılığı ile gerekli emirleri alır ve tanımlı yapılan Pos sistemlerinin üzerinde containerların sağlıklı bir şekilde çalışmasını garanti eder.

- Kube-proxy:

Node lar üzerinde ki ağ kurallarını yönetirler. Bu ağ yapısı cluster içinde ya da dışında olabilir. Pod larımıza yapılacak olan ağ oturumlarının iletişiminden sorumludurlar.

- Kubectl Kurulumu:

Önemli Not: Kubernetes kurulumların yapmadan önce local de kubernetes çalıştırılacak ise ortamda bir Docker Desktop kurulu olması faydalı olacaktır.

<https://kubernetes.io/docs/tasks/tools/> adresinden işletim sistemine göre gerekli kurumların yapılması gereklidir.

- Windows: <https://dl.k8s.io/release/v1.23.0/bin/windows/amd64/kubectl.exe>
- MacOS: <https://kubernetes.io/docs/tasks/tools/install-kubectl-macos/#install-with-homebrew-on-macos> (adresinden brew, curl ile)
- Linux: <https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/#install-kubectl-binary-with-curl-on-linux> (adresinden curl, apt ile)

- Kubernetes Kurulumları:

Önemli Not: Kişisel kullanım, Test ve Geliştirme için Minikube tool ile kurulum yaparak kullanmak ilk defa kullananlar için faydalı olacaktır. Ayrıca Bulut sistemlerin sunduğu servislerde vardır. Bunları ilerleyen zamanlarda göreceğiz. Azure Kubernetes Service(AKS) – Amazon EKS – Google Kubernetes Engine

1- **Docker Desktop:** üzerinden kubernetes kullanılabilir.

Yandaki şekilde görüldüğü üzere docker desktop açılarak Ayarlar->Kubernetes Tıklanır ardından Enable Kubernetes Seçilerek Apply & Restart denilerek aktif edilir.

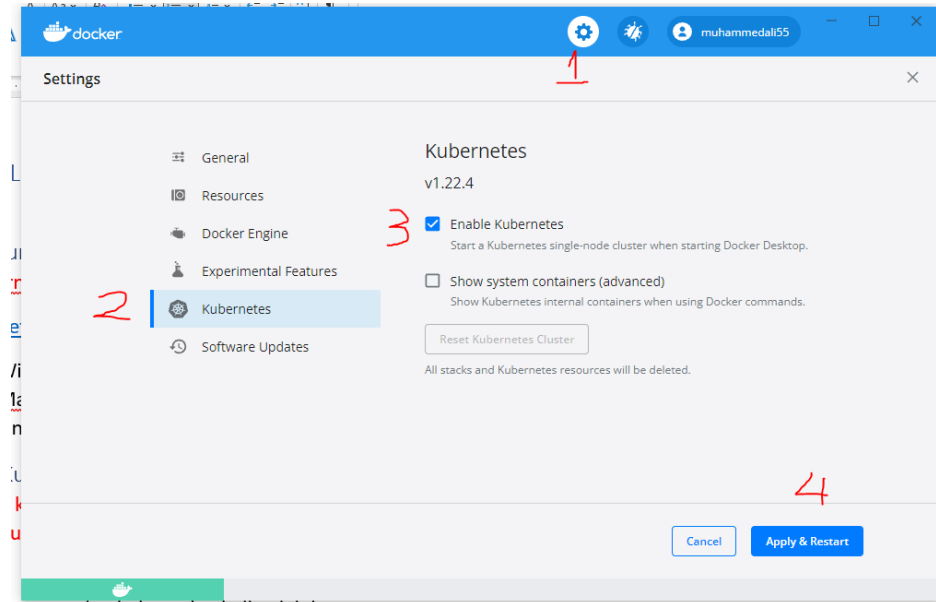
Sitemin çalıştığı;

Power Shell de yazılacak olan

➤ Kubectl get nodes

Komutu ile test edilebilir. Burada docker-desktop un

Çalıştığı grülebilecektir.



2- **Minikube:** <https://minikube.sigs.k8s.io/docs/start/> adresinden işletim sistemine göre kurulumu yapılır. Ardından windows ta PowerShell (Admin modunda) açılarak

➤ **Minikube start**

Komutu çalıştırılır ve sistem kurulumun yapılması sağlanır. Ardından test etmek için yine komut satırında “kubectl get nodes” yazılarak minikube un çalıştığı gözlemlenir.

Önemli Not: Eğer sisteminizde birden çok sanallaştırma var ise, istediğiniz kaynağı seçmek için aşağıdaki komut satırlarını kullanınız.

- `Minikube start --driver=hyper-v`
- `Minikube start --driver=docker`
- `Minikube start --driver=virtualbox`

KUBECTL CONFIGURASYONLARI

Kubectl in config dosyası (C:\Users\MuhammetAli\.kube) şeklindeki bir klasör içinde adı (config) olan bir dosyadır. Bu dosyada cluster yapıları, varsa tüm kubernetes bağlantı ayarlarını barındırır. Hangi kullanıcı hangi alanlara bağlanacak bilgilerin barındırır.

Kubectl yönetim aracı bağlanacağı Kubernetes cluster lara config dosyası aracılığı ile ulaşır. Config dosyasına bağlantı bilgilerini ve bağlanırken kullanmak istediğimiz kullanıcı bilgilerini belirtiriz. Bu bilgileri kullanarak birden fazla namespace oluşturabilir ve contextlerini yaratabiliriz.

Bu yapılandırma dosyasının aracıları ile ilk öğrenmemiz gereken komutlar şöyledir.

- `Kubectl config get-contexts`, var olan tüm bağlantıları gösterir.
- `Kubectl config current-contexts`, aktif olan context adını verir.
- `Kubectl config use-context docker-desktop`, aktif context i istenilen context ile değiştirir.

KUBECTL KOMUTLARINA GÖZATALIM

- `Kubectl get --help`, burada önemli olan help parametresinin kullanılması çünkü tüm komutları ezberlemek imkansızdır. Bu nedenle komutları kullanmak istediğinizde yardım alın.
Önemli Not: ilk olarak kubectl yazılır, ardından bir fiil işlem yapacak kelime gelir (get gibi) sonra çalışacak komutun hangi türden bir nesne üzerinde çalışacağını belirtiyoruz(pods, nodes gibi).opsiyonel olarakte üzerinde çalışacağımız objenin özel adını giriyoruz.
- `Kubectl cluster-info`, cluster ile ilgili bilgileri verir.
- `Kubectl get pods`, aktif namespace üzerinde çalışır ve tüm pod ları listeler
- `Kubectl get pods -n docker-desktop`, şeklinde yazarsam bu docker üzerinde ki pod ların listesini dönecektir.
- `Kubectl get pods -A (--all-namespaces=false) – (--all-namespaces)`, şeklinde ki kullanım ise tüm namespace lerde var olan pod ları listeyecektir.
Not: Default değer true
- `Kubectl get pods -A -o wide (yaml , json)`, şeklindeki kullanım tüm podları geniş şekilde detayları ile gösterir. Yaml ve json olarak ta alabilirsiniz.
- `Kubectl get pods -o json jq -r “.items[].spec.container[].name”`, jq aracını kullanarak sadece isimleri listeyebilirsiniz.(NOT: Yeni kullınımlarına bakılacak.)
- `Kubectl explain pod`, pod objesinin ne olduğu ve hangi alanlarının olduğu bilgilerini döner.

- Pod:

En küçük objemiz pod dur. Podlar bir ya da zorunlu olduğu durumlarda birden fazla container barındırabilirler. Ama genel tek kullanılır. Her pod un benzersiz bir uid si vardır. Her pod benzersiz bir ip adresi barındırır. Aynı pod üzerinde var olan containerlar aynı nodes üzerinde çalıştırılır ve bir birleri ile localhost üzerinden haberleşirler.

- Pos Oluşturmak:

- Kubectl run bilgepod --image=nginx --restart=Never, adı bilgepod olan içerisinde nginx imajını barındıran ve hata durumunda tekrar çalıştırılmamasını istediğim bir pod oluturuyorum. Pod ile ilgili alternatif kodlar;

Test Et(kubectl get pods -o wide).

Geniş bilgi için(kubectl describe pods [podname]).

Pod a ait log bilgisi için(kubectl logs [podname]).

Eğer canlı bir şekilde izlemek istersen(kubectl logs -f [podname])

Pod üzerinde komut çalıştırmak istersen(kubectl exec firstpod -- hostname)

Pod üzerine bağlanmak istersen(kubectl exec -it [podname] -- bash) birkaç komut dene -> ls , printenv, hostname v.s.

Pod u silmek istersen (kubectl delete pods [podname])

- YML Dosyası ile çalışmak:

Pod ve diğer objeleri oluşturmak için en mantıklı yöntem YML dosyaları ile çalışmaktır. Bu nedenle, pod oluştururken yml kullanmak her zaman daha faydalı olacaktır. Bir pod dosyası şuna benzer; →

Bir yml dosyasını kubernate declare etmek için şu komutları kullanırız;

- Kubectl apply -f podcreate.yml

YML dosyaları ile çalışmanın güzel tarafı, podlarda bir değişiklik yapacağımız zaman sadece değiştirmek istediğimiz ya da eklemek istediğimiz parametreleri giriyor ve yeniden apply komutunu kullanıyoruz.

```
podcreate.yml 1 ●
podcreate.yml > {} spec > [ ] containers > {} 0 > [ ]
io.k8s.api.core.v1.Pod (v1@pod.json)
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: ilkpod
5    labels:
6      name: front-end
7  spec:
8    containers:
9      - name: ilkpodname
10        image: nginx:latest
11        ports:
12          - containerPort: 80
```

- Bir Pod un Yaşam Döngüsünü İzlemek

Gerçekte olmayan bir imaj dosyasını eklemek istersek, sistem bu pod u ayağa kaldıramayacak ve reset atacaktır, bir yerden sonra sistem imajın ulaşılabilir olduğunu görünce sistemi hataya çekerek, belli aralıklarla tekrar tekrar deneme yapacaktır. Bu deneme süresi aralıklı artarak devam edecektir. Hadi deneyelim;

- Olmayan bir image ile pod oluştur. kubectl apply -f pod.yml
- Sistemi izlemek için -> kubectl get pods -w (izle komutudur.)
- İzleme ekranında önce hataya sonra, imagepullbackoff a geçtiği görülecektir.
- Eğer spec: altına restartPolicy: Never eklersen, image olmadığı için pod tekrar oluşturulmayacaktır.

- Label ve Selector

Sistemimizde bir çok alanada görev yapan microservis yapısında, developer, test, front-end v.s. bir çok pod olabilir. Bunların yönetimi ve gruplanması önemlidir. Bu nedenle label yapısı çok iyi kurgulanarak kullanılmalıdır. Anahtar ve Değer alanı olarak çalışırlar, dikkat etmemiz gereken konu 63 karakterden uzun isim oluşturmamaktır.

Yanda görüldüğü üzere birden çok pod var ve belli podlar belli takımlar için isimlendirilmiş. Eğer tüm podları label ile görüntülemek isterseniz.

- ➔ Kubectl get pods -l "group" --show-labels
- ➔ Kubectl get pods -l "group=team1" --show-labels

- Annotations

Pod objelerimize özel olarak eklemek istediğimiz açıklamaları bildirmek için kullanırız. Label olarak eklenmesi sakıncalı bilgiler burada tanımlanır.

```
podcreate.yml 9
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: ilkpod
5   labels:
6     name: front-end
7     group: team1
8   annotations:
9     owner: "Muhammet Ali KAYA"
10    email: "muhammedali55@gmail.com"
11    createAt: "07.02.2022"
12 spec:
13   restartPolicy: Never
14   containers:
15   - name: ilkpodname
16     image: nginx:latest
17     ports:
18     - containerPort: 80
```

```
v1@pod+v1@pod+v1@pod.json | v1@pod
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: ilkpod
5   labels:
6     name: front-end
7     group: team1
8 spec:
9   restartPolicy: Never
10  containers:
11  - name: ilkpodname
12    image: nginx:latest
13    ports:
14    - containerPort: 80
15  ---
16  apiVersion: v1
17  kind: Pod
18  metadata:
19    name: ikincipod
20    labels:
21      name: front-end
22      group: team1
23  spec:
24    restartPolicy: Never
25    containers:
26    - name: ilkpodname
27      image: nginx:latest
28      ports:
29      - containerPort: 80
30  ---
31  apiVersion: v1
```

- Namespace

Bu yapı temel olarak, file sistem kısmıdır. Sitemin işleme mantığını anlamak için şöyle bir kurgumuz olsun; bir firmada IT olarak çalışıyorum. Tüm ekiplerin erişebileceği bir dosya sistemine ihtiyacım var bende bir file server kurdum ve herkese bunu açtım. Sistem bir süre iyi çalışır. Sonra herkesin tek bir alana dosyalama yapması nedeniyle yüzlerce binlerce dosya oluşur ve bunları yönetmek güvenliğini sağlamak zorlaşır. Aynı isimde dosya oluşturulamaz, oluştursa benim dosyama ezebilir. V.s.

Bunu çözmenin kolay yolu. Her ekip için ayrı bir klasör oluşturmak ve gruplara yetki vermek yerelidir. İşte tam burada kubernetes içinde de biz namespace ile klasör oluşturabilir, kota, izin gibi ayarları yaparak sistemi işler hale getirebiliriz.

- `kubectl get namespaces`, komutu sistemde ki namespace leri listeler. İlk kurulumda 4 adet namespace vardır. Özellikle bir namespace belirtilmedikçe tüm nesneler default-namespace altında oluşturulur.

Mesela farklı bir namespace altında var olan podları görüntülemek istesek, `kubectl get pods --namespace kube-system` komutunu kullanırız.

YML dosyası ile buna bir örnek verelim;

Yml dosyasından pod oluşturulduktan sonra

- `kubectl get pods` dersiniz oluşturduğunuz pod u göremezsiniz. Sebebi ise siz default namespace içindesiniz. Bu nedenle sorgunuz şöyle olmalı
- `kubectl get pods -n develop-name`
- Ayrıca kod çalıştırma v.s işlemlerinden önce de bunu belirtmelisiniz.
- `kubectl exec -it [podname] -n develop-name -- bash`
- Eğer varsayılan namespace i değiştirmek istiyorsanız o zaman
- `kubectl config set-context --current --namespace=developer-backend`

```
1  apiVersion: v1
2  kind: Namespace
3  metadata:
4    name: develop-name
5  ---
6  apiVersion: v1
7  kind: Pod
8  metadata:
9    namespace: develop-name
10   name: ilkpod
11   labels:
12     name: front-end
13     group: team1
14   annotations:
15     owner: "Muhammet Ali KAYA"
16     email: "muhammedali55@gmail.com"
17     craeteAt: "07.02.2022"
18  spec:
19    restartPolicy: Never
20    containers:
21      - name: ilkpodname
22        image: nginx:latest
23        ports:
24          - containerPort: 80
```

- Deployment:

Burada deployment objesinin kavramak için senaryolarımızı değerlendirelim.

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

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

Gerek yok ama elle oluturmak isterseniz;

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

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

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

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

Test etmek için;

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

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


- Service

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

Types:

ClusterIP, localde çalışır. (Cluster-ip tabanlıdır) NodePort, localde çalışır.(Cluster-ip tabanlıdır)

LoadBalancer, Cloudde ta çalışır.

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

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

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

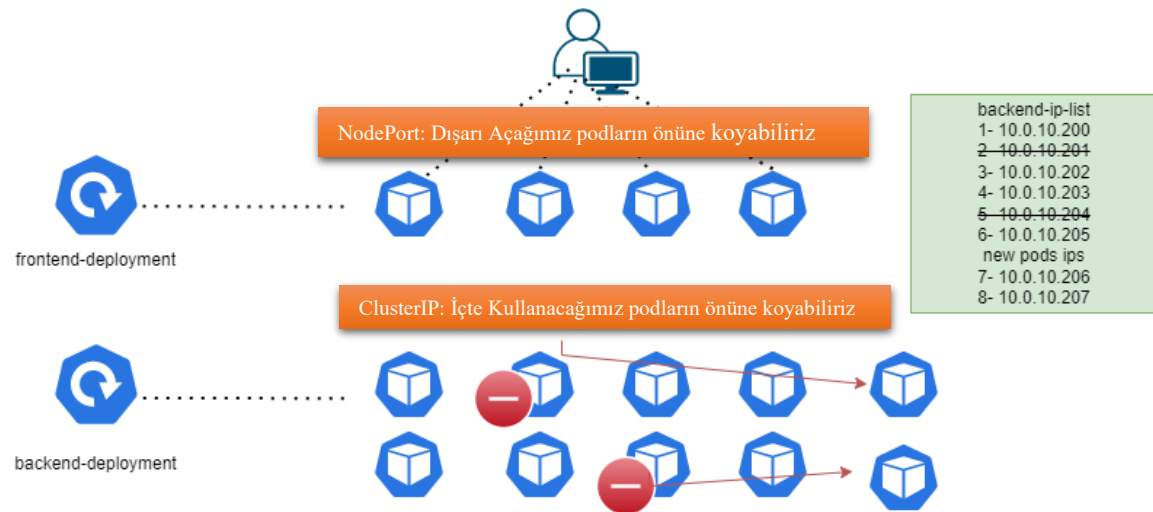
Burada Service test etmek için;

Herhangi bir pod a bağlanıp, servis e aip ip adresi üzerinde istek atabilirsiniz.

nslookup backend ile erişim sağlayabiliyor olmalısınız.

curl backend:8091

- Kubectl get endpoints
- Kubectl describe endpoints frontend



- Liveness probes

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

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

Diyelim ki yakta olması gereken bir DB niz var. Bunu kontrol etmek için belli bir portu dinlemeniz gerekli bunun için [tcpSocket](#) kullanırsınız.

```
io.k8s.api.core.v1.Pod (v1@pod.json) | io.k8s.api.core.v1.Pod
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myapp
5    labels:
6      name: myapp
7  spec:
8    containers:
9      - name: myapp
10       image: k8s.gcr.io/liveness
11       args:
12         - /server
13       livenessProbe:
14         httpGet:
15           path: /healthz
16           port: 8080
17           httpHeaders:
18             - name: Custom-Header
19               value: Awesome
20         initialDelaySeconds: 3
21         periodSeconds: 3
22       resources:
23         limits:
24           memory: "128Mi"
25           cpu: "500m"
26 ---
```

```
io.k8s.api.core.v1.Pod (v1@pod.json) | io.k8s.api.core.v1.Pod
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: myappDB
5    labels:
6      name: myappDB
7  spec:
8    containers:
9      - name: myappDB
10       image: k8s.gcr.io/goproxy:0.1
11       ports:
12         - containerPort: 8080
13       livenessProbe:
14         tcpSocket:
15           port: 8080
16         initialDelaySeconds: 15
17         periodSeconds: 20
18       resources:
19         limits:
20           memory: "128Mi"
21           cpu: "500m"
22 ---
```

- Resources

Kaynakları sınırlı kullanmak için ihtiyacımızı giderir.

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

- Environment Variable

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

➤ `Kubectl exec [podname] -- printenv`

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

- Secret

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

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

Secretleri görmek için;

- Kubectl get secrets
- Kubectl describe secrets mytoolssecret

```
podcreate.yml > {} spec
io.k8s.api.core.v1.Secret (v1@secret.json) | v1@secret+v10
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: mytoolssecret
5    type: Opaque
6  stringData:
7    db_mongourl: "192.1.1.2"
8    db_username: "admin"
9    db_password: "$İffffreeeee"
10 ---
11 apiVersion: v1
12 kind: Pod
13 metadata:
14   name: myapp
15 labels:
16   name: myapp
17 spec:
18   containers:
19   - name: myapp
20     image: <Image>
21     resources:
22     limits:
23       memory: "128Mi"
24       cpu: "500m"
25     ports:
26     - containerPort: 80
27     env:
28     - name: MONGOURL
29       valueFrom:
30       secretKeyRef:
31         name: mytoolssecret
32         key: db_mongourl
33
```

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

- ConfigMap:

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

```
10  apiVersion: v1
11  kind: Pod
12  metadata:
13    name: configmappod
14  spec:
15    containers:
16    - name: configmappod
17      image: muhammedali55/XXXX
18      resources:
19        limits:
20          memory: "128Mi"
21          cpu: "500m"
22      env:
23        - name: MONGO_URL_POD
24          valueFrom:
25            configMapKeyRef:
26              name: bilgevonfigmap
27              key: MONGO_URL
28        - name: MONGO_ADMIN_POD
29          valueFrom:
30            configMapKeyRef:
31              name: bilgevonfigmap
32              key: MONGO_ADMIN
33    ports:
34    - containerPort: 80
35
```

```
v1@configmap+v1@pod.json | v1@configm
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: bilgevonfigmap
5  data:
6    MONGO_URL: "localhost"
7    MONGO_ADMIN: "root"
8    MONGO_PASSWORD: "root"
9  ---
```

- Volume

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

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

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: volumepath
5    labels:
6      name: volumepath
7  spec:
8    containers:
9      - name: volumacontainer
10        image: muhammedali55/tinyimagevolume
11        volumeMounts:
12          - name: directory-volume
13            mountPath: /logs
14          - name: file-volume
15            mountPath: /logs/alllogs.log
16        resources:
17          limits:
18            memory: "128Mi"
19            cpu: "500m"
20        volumes:
21          - name: directory-volume # Zaten container üzerinde var olan bir dizini belirtir.
22            hostPath:
23              path: /logs
24              type: Directory
25          - name: dircreative-volume # bir dizin oluştur ancak eğer bu dizin zaten container üzerinde var ise oluşturma
26            hostPath:
27              path: /temp
28              type: DirectoryOrCreate
29          - name: file-volume # Eğer alllogs.log dosyası yok ise oluştur diyoruz.
30            hostPath:
31              path: /logs/alllogs.log
32              type: FileOrCreate
33
```

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

- Node Affinity:

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

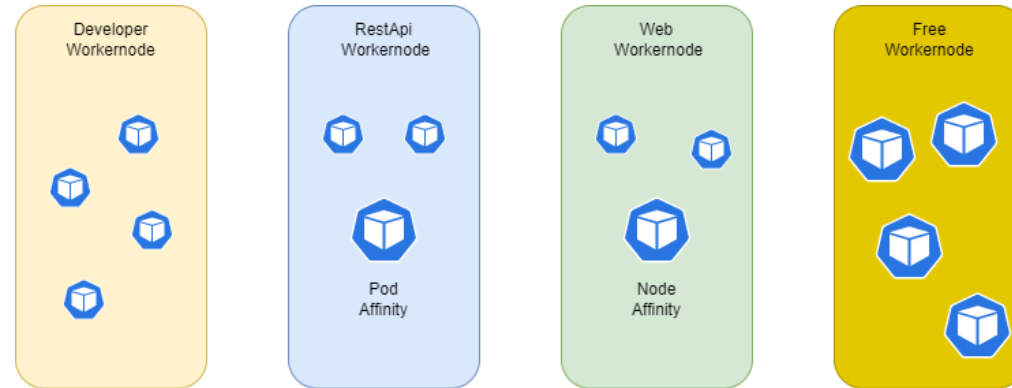
- Pod Affinity:

Podların çalışacağı nod gruplarını belirleyebiliriz.

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

- Taint ve Toleration:

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



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

Öncelikle sistemimizde çalışan workernode lara bir göz atalım

➤ `Kubectl describe nodes minikube ya da docker-desktop`

Peki bir node üzerine nasıl ekleme yapılabilir?

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

➤ `Kubectl taint node minikube ya da docker-desktop developerteam- TAİN SİLME(sonuna – işareti konulur.)`

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

Oluşturma işleminden sonra, bunu denemek ve test etmek için sisteminizde ki podları izlemeye başlayın

➤ `Kubectl get pods -w`

Ardından yeni bir pod oluşturmayı deneyin.

➤ `Kubectl run deneme --image=nginx --restart=Never`

Bu işlemden sonra podları gözlemleyin. Bu işlemde pod üzerinde bir taint eklediğim için node bu pod u tolere edemedi ve pod “pending” te takılı kaldı.

➤ `Kubectl describe pod deneme`

Yukarıdaki komutu çalıştırıp detaylara bakarsanız sorunu burada görebilirsiniz.

```
v1@pod+v1@pod.json | v1@pod+v1@pod.json
apiVersion: v1
kind: Pod
metadata:
  name: tolerepod
  labels:
    name: develop
spec:
  containers:
  - name: tolerepod
    image: muhammedali55/tinyimagevolume
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
  tolerations: # node tolere edebilmesi için tüm parametreler node ile ayın girildi
  - key: "developerteam"
    operator: "Equal"
    value: "bilgeadam"
    effect: "NoSchedule"
```

```
apiVersion: v1
kind: Pod
metadata:
  name: tolerepod2
  labels:
    name: developertest
spec:
  containers:
  - name: tolerepod2
    image: muhammedali55/tinyimagevolume
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
  tolerations: # value yok, eğer developerteam bir key varsa ve NoSchedule ise tolere et
  - key: "developerteam"
    operator: "Exists"
    effect: "NoSchedule"
```

ÇOOK ÖEMLİ: Eğer bir pod üzerinde çalışamayacağı bir node üzerinde oluşturulursa orada kalmaya devam eder. Ne çalışır ne de silinir. İşte burada tolere edilmeyen pod ların silinmesi gereklidir bu nedenle taint yazarken şunu yazmak daha mantıklıdır

➤ `Kubectl taint node minikube developerteam=bilgeadam:NoExecute`

- **DaemonSet:**

Sorun: diyelimki bir 100 worker node tan oluşan bir kubernetes cluster yapınız var. Ve bunların tamamı üzerinde çalışması gereken bir uygulama yapımız olsun. Amacımız uygulamamızda toplayacağımız hata, işlem v.s. loglarımızı merkezi bir yerde toplamak istiyoruz. Şimdi tüm node larda çalışacak ve topladığı log kayıtlarını aktaracak bir uygulamamız olsun.

Burada sorunu şöyle çözemeyi deneyebiliriz. Uygulamamızı tüm node lara tektek bağlanarak kurulum yapabiliriz.

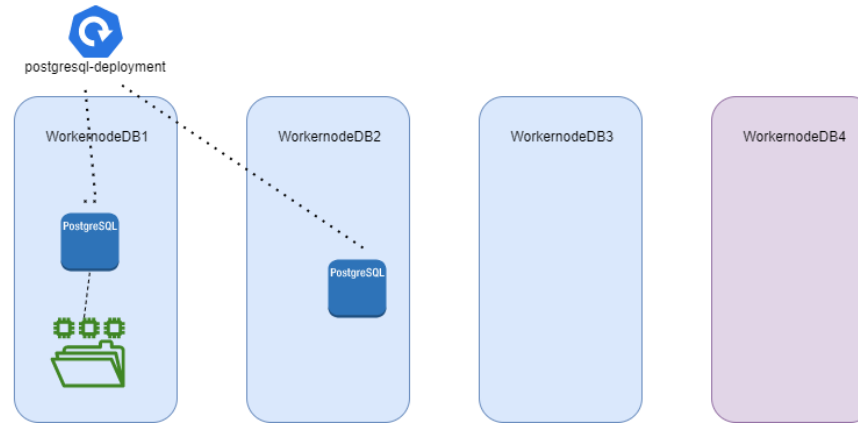
Ya da uygulamamı container haline getirip 100 adet yml dosyayı oluşturarak yükleyebilirim. 😊

İşte tüm bu sorunların ortasında, daemonset objesi, bize yönetilebilir bir node yönetimi sunar. Böylece sisteme node eklendikçe üzerinde çalıştırılacak podlarda eklenir.

Not: Şu aşamada sizler için gerekli bir obje değil bu nedenle örnek bir yapıyı ekledim.

- **Persistent Volume**

Sorun: Diyelim ki ölçeklenebilir bir sistem ile postgresql gibi bir Veritabanını farklı node lar üzerinde çalıştırmak isteyelim. Eğer bir node üzerinde çalışan uygulamada eğer workernode kapanırsa , deployment objesi uygun bir node bularak bu uygulamayı farklı bir node üzerinde çalıştırır. Ancak burada sorun şu tüm veriler diğer node üzerinde kaldı 😞



NOT: Kubernetes hali hazırda üzerinde bir çok bulut sistemde var olan dosya sistemleri ile uyumlu şekilde çalışabilecek driver ları içerir. Ancak eğer 3. Parti bir yazılım ile işlemler yapacaksanız bu durumda Container Storage Interface(CSI) kullanarak işlemlerinizi yapabilirsiniz.

İlk olarak bir Persistence Volume yaratmamız gereklidir. Data sonra bunu podlara bağlayabilmek için PersistentVolumeClaim objesi oluşturmamız gerekir.

```
1 io.k8s.api.core.v1.PersistentVolume (v1@persistentvolume.json)
2 apiVersion: v1
3 kind: PersistentVolume
4 metadata:
5   name: postgresqldb
6   labels:
7     app: postgres
8 spec:
9   capacity:
10     storage: 5Gi
11   accessModes:
12     - ReadWriteOnce #ReadWriteOnce-> Bu volume aynı anda sadece bir pod a bağlanabilir. Read-Write
13     #ReadOnly-> Bu volume aynı anda birden fazla pod a bağlanabilir. Read- NonWrite
14     #ReadWrite-> birden fazla pod a bağlanabilir. Read-Write
15   persistentVolumeReclaimPolicy: Recycle
16   nfs:
17     path: /db
18     server: 172.17.0.2
```

```
1 io.k8s.api.core.v1.PersistentVolumeClaim (v1@persistentvolumeclaim.json)
2 apiVersion: v1
3 kind: PersistentVolumeClaim
4 metadata:
5   name: postgresqlbclaim
6 spec:
7   resources:
8     requests:
9       storage: 5Gi
10   volumeMode: Filesystem
11   accessModes:
12     - ReadWriteOnce
13   storageClassName: ""
14   selector:
15     matchLabels:
16       app: postgres
```

```
1 io.k8s.api.apps.v1.Deployment (v1@deployment.json)
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: postgresqldeployment
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10       app: postgresql
11   template:
12     metadata:
13       labels:
14         app: postgresql
15     spec:
16       containers:
17         - name: postgresql
18           image: postgres
19           ports:
20             - containerPort: 5432
21           volumeMounts:
22             - mountPath: "/var/lib/postgres"
23               name: myvolume
24         - name: myvolume
25           persistentVolumeClaim:
26             claimName: postgresqlbclaim
```

- `docker volume create nfsvol`
- `docker network create --driver=bridge --subnet=10.255.255.0/24 --ip-range=10.255.255.0/24 --gateway=10.255.255.10 nfsnet`
- `docker run -dit --privileged --restart unless-stopped -e SHARED_DIRECTORY=/data -v nfsvol:/data --network nfsnet -p 2049:2049 --name nfssrv muhammedali/basicnfs`

Yukarıdaki ayarları kullanarak, minikube üzerinde bir NFS ayağa kaldırabilir ve bunun üzerinden volüme işlemleri yapabilirsiniz. İşlemler karmaşık ve sıkıntılı olduğu için bu kısımları geçiyoruz. Genellikle developer olarak bu kısımlar bizi çokta ilgilendirmiyor.

- StatefulSet:

Sorun: düşününki deployment objesi ile podlar oluşturuyorsunuz. Sisteminiz genişletmek istediğiniz de replikaları arttırarak sistemin genişlemesini sağlayabilirsiniz. Burada deployment replika setlerini rast gele genişletir ve siler. İşte tam burada NoSql çözümlerinin genişlemelerinde sorun yaşanabilir. Nasıl yani? Bir nosql çözümünde master -> slave + slave + slave şeklinde çalışır. Biz bir pod u master yaptık diyelim, diğerlerini secont olsun. Bu durumda bizim sistemimiz önce ihtiyaçtan büyüdü ve 5 pod daha ekledik, sonra sistem ihtiyaç duymayacak hale geldi ve pod sayısını eski haline getirmek istedik. Ancak deployment pod ları silerken master pod u da **sildi sistem patladı** 😞

- Job:

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

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

Sistemi izlemek için

- Kubectl get pods -w
- Kubectl get jobs -w

EEEEEEET peki job işlemlerinin belli bir zaman diliminde çalışmasını istese idik ne yapacaktık?

- CronJob

İşlerin belli bir zaman diliminde yapılmasını sağlamak için kullanılır.

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