CKA

Persistant volume and claim and storage class

1. What is the size of PV is required
2. What is the mount point
3. What is the permission(read/write) capability for it
4. what is reclaim(retain or delete after pod deletion) policy
   K8s administrator going to create volume based on above 3 points from where he need to create: NFS, EBS, Ceinder, Cyph storage
5. Admin creates PV and user creates pvc based on request and properties set on volume

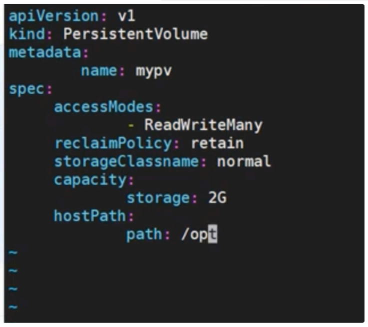
   there is one is to one relationship between pv and pvc and if two volumes present satisfying pvc requirements (capacity, access-modes, storage-class, volume modes) and if you wanna choose specific volume then you can use labels and selector to bind to right volume

   Finally, note that a smaller claim may get bound to a larger volume if all the other criteria matches, and there are no better options. There is a one to one relationship between claims and volumes, so no other claims can utilize the remaining capacity in the volume. If there are no volumes available, the persistent volume claim will remain in a pending state until newer volumes are made available to the cluster.

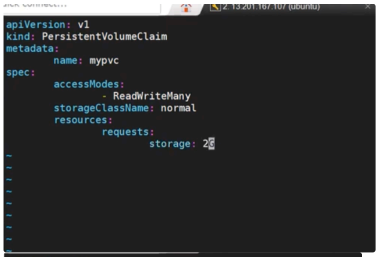
   Use the following manifest file to create a `pv-log` persistent volume:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-log
spec:
  persistentVolumeReclaimPolicy: Retain
  accessModes:
    - ReadWriteMany
  capacity:
    storage: 100Mi
  hostPath:
    path: /pv/log

```

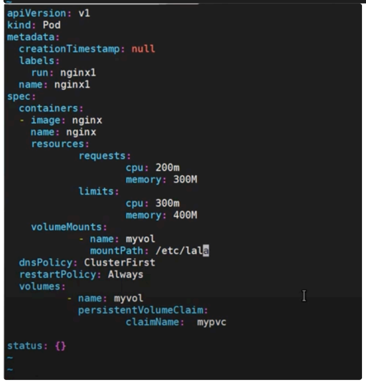
   Then run the command `kubectl create -f <file-name>.yaml` to create a PV from manifest file.



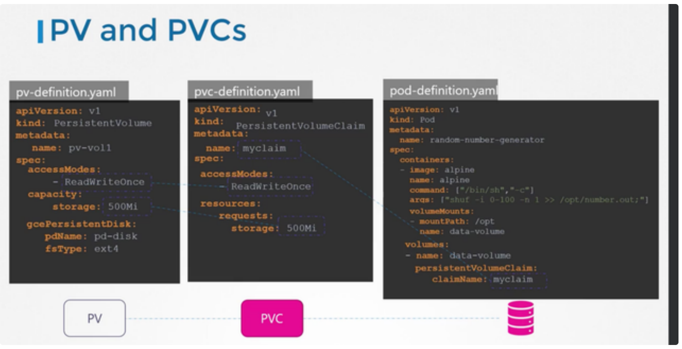
```
apiVersion: v1
kind: PersistentVolume
metadata:
        name: mypv
spec:
      accessModes:
              - ReadWriteMany
      reclaimPolicy: retain
      storageClassname: normal
      capacity:
              storage: 2G
      hostPath:
              path: /opt
~
~
~
~
```



```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
        name: mypvc
spec:
        accessModes:
                - ReadWriteMany
        storageClassName: normal
        resources:
                requests:
                        storage: 2G
~
~
~
~
~
~
~
~
```



```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx1
  name: nginx1
spec:
  containers:
  - image: nginx
    name: nginx
    resources:
            requests:
                    cpu: 200m
                    memory: 300M
            limits:
                    cpu: 300m
                    memory: 400M
    volumeMounts:
            - name: myvol
              mountPath: /etc/lala
  dnsPolicy: ClusterFirst
  restartPolicy: Always
  volumes:
          - name: myvol
            persistentVolumeClaim:
                    claimName:  mypvc

status: {}
~
~
```



PV and PVCs

pv-definition.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  accessModes:
      - ReadWriteOnce
  capacity:
      storage: 500Mi
  gcePersistentDisk:
     pdName: pd-disk
     fsType: ext4
```

pvc-definition.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
      - ReadWriteOnce
  resources:
     requests:
        storage: 500Mi
```

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh","-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
    volumeMounts:
    - mountPath: /opt
      name: data-volume
  volumes:
  - name: data-volume
    persistentVolumeClaim:
      claimName: myclaim
```

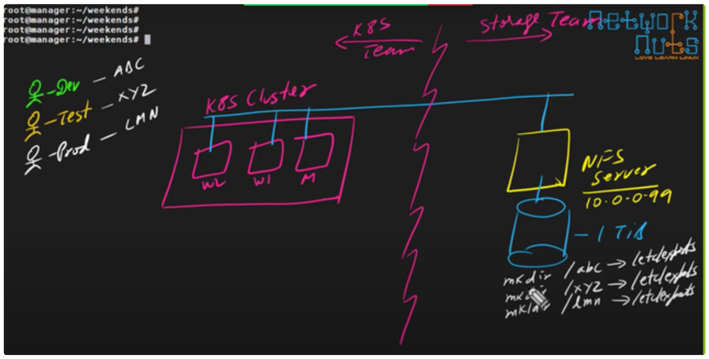
6.

if we want to release PV, first ensure associated pod is deleted and then check the PVC if that is deleted, later PV will get release

there are 2 ways of creating PV---

Static -in static whenever dev team requested to create volume so manually need to create PV with their requirements and then we can claim by PVC.

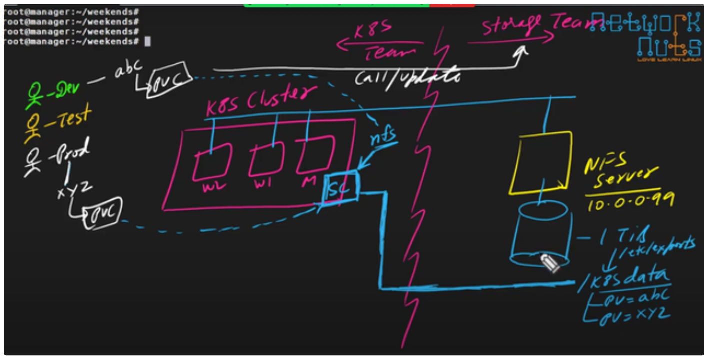
The problem here is that before this PV is created, you must have created the disk on Google Cloud. Every time an application requires storage, you have to first manually provision the disk on Google Cloud, and then manually create a persistent volume definition file using the same name as that of the disk that you created. That's called static provisioning volumes.



```
root@manager:~/weekends#
root@manager:~/weekends#
root@manager:~/weekends#
root@manager:~/weekends#
```

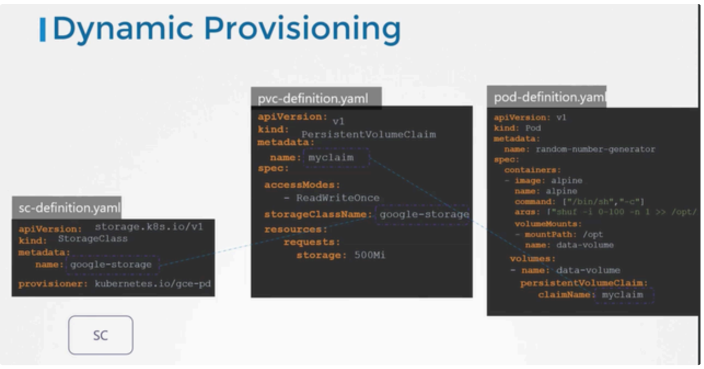
Dynamic-in dynamic case we are just writing the PVC based, then we will be attaching to Storage class storage class will automatically create PV.

With storage classes, you can define a provisioner, such as Google Storage, that can automatically provision storage on Google Cloud and attach that to pods when a claim is made.That's called dynamic provisioning of volume.



```
root@manager:~/weekends#
root@manager:~/weekends#
root@manager:~/weekends#
root@manager:~/weekends#
```

You do that by creating a storage class object with the API version set to storage.k8.io/v1,specify a name, and use **provisioner as Kubernetes.io/gce-pd.** So going back to our original state where we have a pod using a PVC for its storage, and the PVC is bound to a PV, we now have a storage class,so **we no longer need the PV definition,** because the PV and any associated storage is going to be created automatically when the storage class is created.For the PVC to use the storage class we defined, we specify the storage class name in the PVC definition.That's how the PVC knows which storage class to use.



Dynamic Provisioning

sc-definition.yaml

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: google-storage

provisioner: kubernetes.io/gce-pd
```

pvc-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
 accessModes:
   - ReadWriteOnce
 storageClassName: google-storage
 resources:
   requests:
     storage: 500Mi
```

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh","-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/
    volumeMounts:
    - mountPath: /opt
      name: data-volume
   volumes:
   - name: data-volume
     persistentVolumeClaim:
       claimName: myclaim
```

- The StorageClass used by the PVC uses `WaitForFirstConsumer` volume binding mode. This means that the persistent volume will not bind to the claim until a pod makes use of the PVC to request storage.
- The Storage Class called `local-storage` makes use of `VolumeBindingMode` set to `WaitForFirstConsumer`. This will delay the binding and provisioning of a PersistentVolume until a Pod using the PersistentVolumeClaim is created.
- The `local-storage` storage class makes use of the no-provisioner and currently does not support dynamic provisioning.

  Refer to the tab above the terminal (called Local Storage) to read more about it.

  Troubleshooting:
  Why is the claim not bound to the available `Persistent Volume`?

Run the command: `kubectl get pv,pvc` and look under the `Access Modes` section.
The Access Modes set on the PV and the PVC do not match.

Why is the PVC stuck in `Terminating` state?

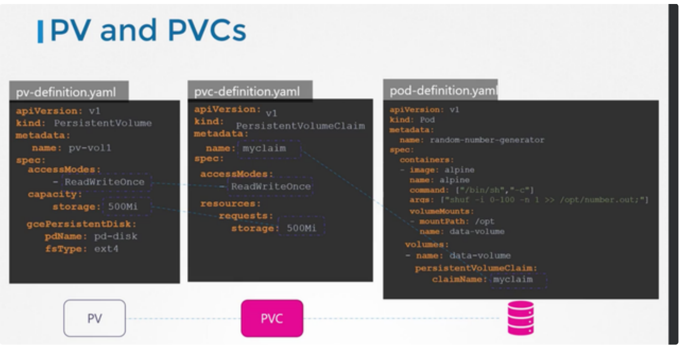
The PVC was still being used by the `webapp` pod when we issued the delete command. Until the pod is deleted, the PVC will remain in a `terminating` state.

Create a new `PersistentVolumeClaim` by the name of `local-pvc` that should bind to the volume `local-pv`.
ans : Inspect the persistent volume and look for the **Access Mode, Storage and StorageClassName** used. Use this information to create the PVC.

ClusterRole is a non-namespaced resource. You can check via the `kubectl api-resources --namespaced=false` command. So the correct answer would be `Cluster Roles are cluster wide and not part of any namespace`.

exam point of view:

For creating PV: accessmode, retain policy, capacity:storage, hostpath
For creating PVC: accessmode, storageclassname, resource:request:storage
For pod: volumeMount are specific to containers and volumes are specific to pod. If 1 pod has 2 containers, then 2 volumemounts and only 1 volume parameter.
Under volumes in pod, we are going attach pv by making use of pvc, hence actual pvc name (claimName)needs to be given. also ensure volumes and volumeMount name should be same.



PV and PVCs

pv-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
   name: pv-vol1
spec:
  accessModes:
      - ReadWriteOnce
  capacity:
      storage: 500Mi
  gcePersistentDisk:
     pdName: pd-disk
     fsType: ext4
```

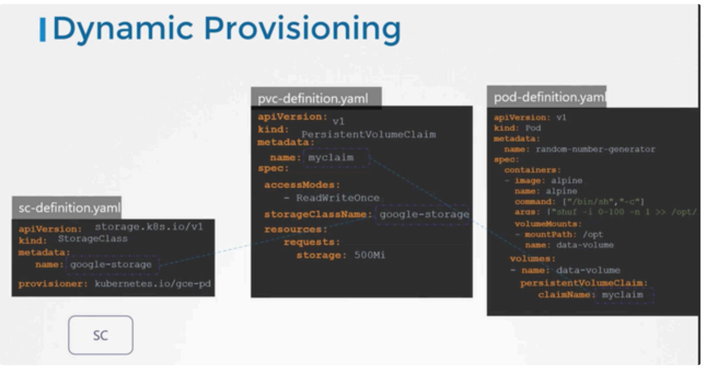
pvc-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
 accessModes:
     - ReadWriteOnce
 resources:
    requests:
      storage: 500Mi
```

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh","-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
    volumeMounts:
    - mountPath: /opt
      name: data-volume
   volumes:
   - name: data-volume
     persistentVolumeClaim:
        claimName: myclaim
```

If pv you are creating manually then in pvc storageClassName value wud be normal, however when it comes to dynamic pv provisioning, in pvc you gonna give storageClassName as actual storage class what we created



Dynamic Provisioning

sc-definition.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: google-storage
provisioner: kubernetes.io/gce-pd
```

pvc-definition.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: google-storage
  resources:
    requests:
      storage: 500Mi
```

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh","-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/
    volumeMounts:
    - mountPath: /opt
      name: data-volume
  volumes:
  - name: data-volume
    persistentVolumeClaim:
      claimName: myclaim
```

Network Policy

If you haven't explicitly applied any network policies to your cluster, the default behavior will typically allow unrestricted communication between pods within the same namespace and across namespaces, as well as with external endpoints.

What is the network interface configured for cluster connectivity on the `controlplane` node?

Run: `kubectl get nodes -o wide` to see the IP address assigned to the `controlplane` node.

```
controlplane:~# kubectl get nodes controlplane -o wide
NAME           STATUS   ROLES           AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION   CONTAINER-RUNTIME
controlplane   Ready    control-plane   7m    v1.29.0   192.23.97.3   <none>        Ubuntu 22.04.3 LTS   5.4.0-1106-gcp   containerd://1.6.26

controlplane:~#
```

In this case, the internal IP address used for node to node communication is `192.23.97.3`.

Important Note : The result above is just an example, the node IP address will vary for each lab.

Next, find the network interface to which this IP is assigned by making use of the `ip a` command:

```
controlplane:~# ip a | grep -B2 192.23.97.3
25556: eth0@if25557: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:c0:17:61:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.23.97.3/24 brd 192.23.97.255 scope global eth0

controlplane:~#
```

Here you can see that the interface associated with this IP is `eth0` on the host.

Run the command: `kubectl get networkpolicy` or `kubectl get netpol`

Drain and corden

Drain : it will evicts the pods from existing but --ignore-daemon-sets. `kubectl drain node-1`

Drain command itself evict the pod and also make that unschedulable.

cordoning: if you want to only unshedulable we an make use of cordon. removing the node from cluster making our node unschedulable and it. `kubectl cordon node-1`

uncoderning: attaching the node to cluster. `kubectl uncordon node-1`

`kubectl drain node01 --ignore-daemonsets`

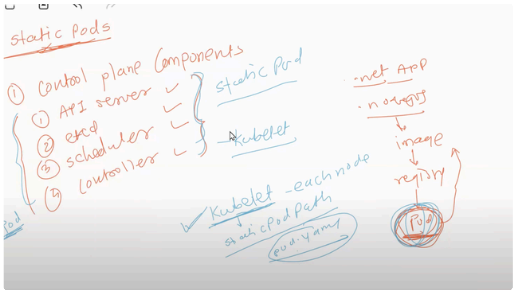
Daemon set

An easy way to create a DaemonSet is to **first generate a YAML file for a Deployment** with the command `kubectl create deployment elasticsearch --image=registry.k8s.io/fluentd-elasticsearch:1.20 -n kube-system --dry-run=client -o yaml > fluentd.yaml`. Next, remove the **replicas, strategy and status fields** from the YAML file using a text editor. Also, change the kind from `Deployment` to `DaemonSet`.

Finally, create the **Daemonset** by running `kubectl create -f fluentd.yaml`

For full info
`Kubectl get Daemonsets -A -o wide`

Static POD



Static pods are essential for managing control plane components and custom pods are managed by control plane components within the Kubernetes infrastructure. The kubelet handles static pod management, with YAML files located in a specific path mentioned on `/var/lib/kubelet/config.yaml` each node. This ensures that all control plane components run effectively as static pods.

- **Static pods will always be associated with node name**
- **From the output we can see that the kubelet config file used is** `/var/lib/kubelet/config.yaml`
- **If you wanna add or delete any new pod as static pod , just make an entry in the same fil**e
- Create a pod definition file in the manifests folder. To do this, run the command:
  `kubectl run --restart=Never --image=busybox static-busybox --dry-run=client -o yaml --command -- sleep 1000 > /etc/kubernetes/manifests/static-busybox.yaml`

QoS

Kubernetes uses QoS classes to make decisions about evicting Pods when Node resources are exceeded.

When Kubernetes creates a Pod it assigns one of these QoS classes to the Pod:

- Guaranteed
- Burstable
- BestEffort

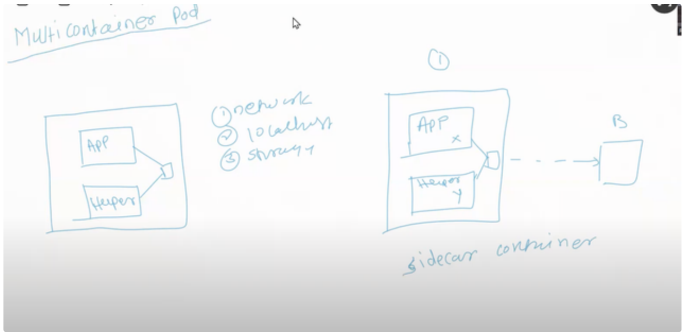
Configure Quality of Service for Pods

Multiple container

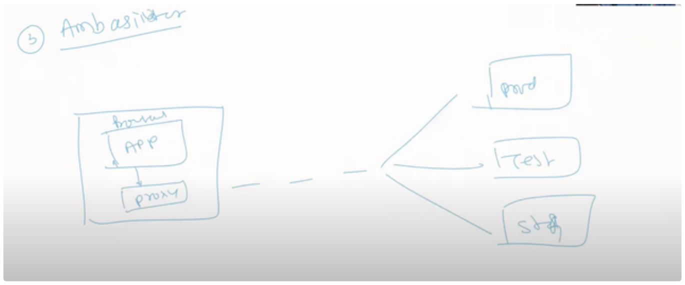
we are running more than one container in same pod

all container will share same ntwork and storage

1. sidecar container: let assume we have application producing logs in x format but user want in y format as we know with multicontainer concept we can run helper container in same pod as well all know they are sharing same storage so helper will read x format from storage and will convert into y as needed.

   Ambasidder:

   instead of exposing our application directly to the external world create a proxy container and through that we can connect to the external DB

Namespaces

- it is a virtual cluster or logical partition in our cluster
- We are unable to utilize the same pod name within the same namespace, but it is feasible to use it in a different namespace within the same cluster.
- if you have not created any namespace it will deploy in default ns
- once we create the cluster by default below ns is going to create
  - kube-system---system related pods are running-- we can't deploy our application pods here
  - kube-public
  - default

  - kube-node-lease

  How to check the specific pod in all namespace. >> `kubectl get pod -A | grep blue`

- What DNS name should the `Blue` application use to access the database `db-service` in the `dev` namespace?

  Use port `6379 >>>>>>` `db-service.dev.svc.cluster.local`.

Since the `blue` application and the `db-service` are in different namespaces. In this case, we need to use the service name along with the namespace to access the database. The `FQDN (fully Qualified Domain Name)` for the `db-service` in this example would be `db-service.dev.svc.cluster.local`.<svc.name><namespace>.svc.cluster.local
`Note:` You can also access it using the service name and namespace like this: `db-service.dev`

Grafana

Loki is utilized for logging, while Prometheus is employed for handling metrics, primarily focusing on time series data.

RBAC

role: it is specific to one **namespace**

rolebinding: binding the role to **specific user**

clusterrole: it is for **multiple namespace** inside same cluster

clusterrolebinding: binding the **clusterrole to specific user**

context: mapping the **user to cluster**

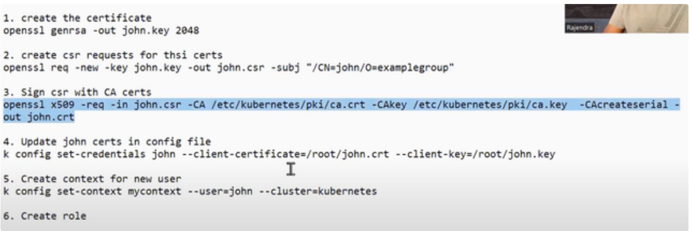
How to create the user?

user creation: key generation ----> csr(certificate sign request)---->CA(get it sign from cluster authority)

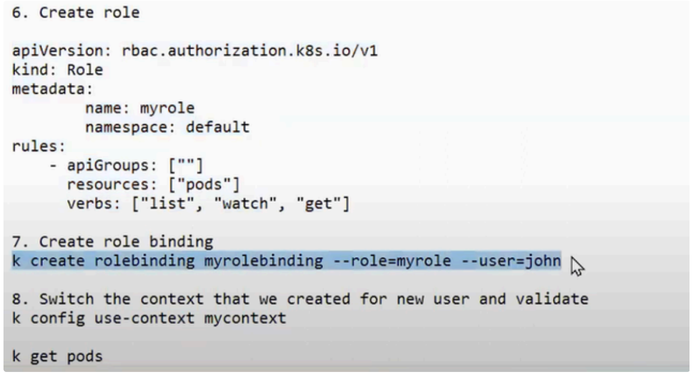
set credentials-----> setting the credentials(key and certs) for the user created in the kubeconfig file(we can see it in kube-config)

set-context----> mapping the user to cluster

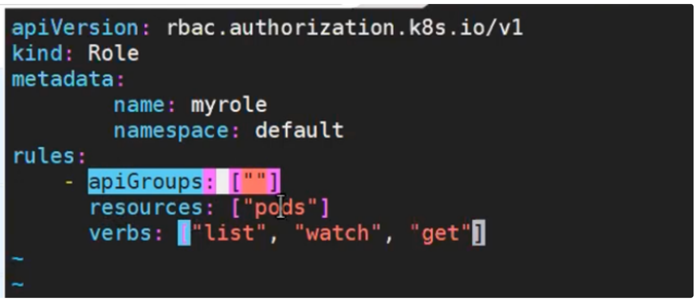
switching into above context---> then it will perform the actions based on role we binded to that user within that context



```
1. create the certificate
openssl genrsa -out john.key 2048

2. create csr requests for thsi certs
openssl req -new -key john.key -out john.csr -subj "/CN=john/O=examplegroup"

3. Sign csr with CA certs
openssl x509 -req -in john.csr -CA /etc/kubernetes/pki/ca.crt -CAkey /etc/kubernetes/pki/ca.key  -CAcreateserial -
out john.crt

4. Update john certs in config file
k config set-credentials john --client-certificate=/root/john.crt --client-key=/root/john.key

5. Create context for new user
k config set-context mycontext --user=john --cluster=kubernetes

6. Create role
```



```
6. Create role

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

7. Create role binding
k create rolebinding myrolebinding --role=myrole --user=john

8. Switch the context that we created for new user and validate
k config use-context mycontext

k get pods
```



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

Identify the ETCD Server CA Root Certificate used to serve ETCD Server.

Note: ETCD can have its own CA. So this may be a different CA certificate than the one used by kube-api server.
ans: Look for CA Certificate (`trusted-ca-file`) in file `/etc/kubernetes/manifests/etcd.yaml` .

que: Kubectl suddenly stops responding to your commands. Check it out! Someone recently modified the `/etc/kubernetes/manifests/etcd.yaml` file

You are asked to investigate and fix the issue. Once you fix the issue wait for sometime for kubectl to respond. Check the logs of the ETCD container.

The certificate file used here is incorrect. It is set to `/etc/kubernetes/pki/etcd/server-certificate.crt` which does not exist. As we saw in the previous questions the correct path should be `/etc/kubernetes/pki/etcd/server.crt` .

```
root@controlplane:~# ls -l /etc/kubernetes/pki/etcd/server* | grep .crt
-rw-r--r-- 1 root root 1188 May 20 00:41 /etc/kubernetes/pki/etcd/server.crt
root@controlplane:~#
```

Update the YAML file with the correct certificate path and wait for the ETCD pod to be recreated. wait for the `kube-apiserver` to get to a `Ready` state.

Identify the key used to authenticate `kubeapi-server` to the `kubelet` server.

Look for `kubelet-client-key` option in the file `/etc/kubernetes/manifests/kube-apiserver.yaml` .

- for certificates and keys of CP components the path: `/etc/kubernetes/pki`
- for etcd remember there is etcd folder `/etc/kubernetes/pki/etcd/`
  Run the command `openssl x509 -in /etc/kubernetes/pki/etcd/server.crt -text` and look for `Subject CN` .
- for static pods: `/etc/kubernetes/manifests/kube-apiserver.yaml`
- ETCD has its own CA. The right CA must be used for the ETCD-CA file in `/etc/kubernetes/manifests/kube-apiserver.yaml`
- Run the command `openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text` and look for Subject CN.

NOTE: wrt to etcd, we have 2 certs, etcd server certs and trusted ca certs

What is the Common Name (CN) configured on the Kube API Server Certificate?

OpenSSL Syntax: `openssl x509 -in file-path.crt -text -noout`

Run the command `openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text` and look for Subject CN.

What is the name of the CA who issued the Kube API Server Certificate?

Run the comman `openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text` and look for issuer.

The kube-api server stopped again! Check it out. Inspect the kube-api server logs and identify the root cause and fix the issue.

Run `crictl ps -a` command to identify the kube-api server container. Run `crictl logs container-id` command to view the logs.If we inspect the `kube-apiserver` container on the `controlplane` , we can see that it is frequently exiting.

```
root@controlplane:~# crictl ps -a | grep kube-apiserver
1fb242055cff8       529072250ccc6       About a minute ago   Exited              kube-apiserver            3                   ed2174865a416       kube-apiserver-
```

If we now inspect the logs of this exited container, we would see the following errors:

```
root@controlplane:~# crictl logs --tail=2 1fb242055cff8
W0916 14:19:44.771920       1 clientconn.go:1331] [core] grpc: addrConn.createTransport failed to connect to {127.0.0.1:2379 127.0.0.1 <nil> 0 <nil>}. Err: connec
E0916 14:19:48.689303       1 run.go:74] "command failed" err="context deadline exceeded"
```

This indicates an issue with the ETCD CA certificate used by the `kube-apiserver` . Correct it to use the file `/etc/kubernetes/pki/etcd/ca.crt` .

Once the YAML file has been saved, wait for the `kube-apiserver` pod to be `Ready` . This can take a couple of minutes.

Create a **CertificateSigningRequest** object with the name `akshay` with the contents of the `akshay.csr` file. As of kubernetes 1.19, the API to use for CSR is `certificates.k8s.io/v1` .

Please note that an additional field called `signerName` should also be added when creating CSR. For client authentication to the API server we will use the built-in signer `kubernetes.io/kube-apiserver-client` .

Use this command to generate the base64 encoded format as following: -

```
cat akshay.csr | base64 -w 0
```

Finally, save the below YAML in a file and create a CSR name `akshay` as follows: -
Certificates and Certificate Signing Requests

```
---
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: akshay
spec:
  groups:
```

```
- system:authenticated
request: <Paste the base64 encoded value of the CSR file>
signerName: kubernetes.io/kube-apiserver-client
usages:
- client auth
```

```
kubectl apply -f akshay-csr.yaml
```

Normal CSR is starts with M-- decoded one we need to encode it `cat myuser.csr | base64 | tr -d "\n"`

Approve the CSR Request: `kubectl certificate approve username`

Reject the CSR Request: `kubectl certificate deny agent-smith`

Hmmm.. You are not aware of a request coming in. What groups is this CSR requesting access to?

Check the details about the request. Preferebly in YAML.

Run the command `kubectl get csr agent-smith -o yaml`

Delete the new CSR object: `kubectl delete csr agent-smith`

command to check the cluster number: `kubectl config view`

apiVersion: v1
clusters:

- cluster:
  certificate-authority-data: DATA+OMITTED
  server: https://controlplane:6443
  name: kubernetes
  contexts:
- context:
  cluster: kubernetes
  user: kubernetes-admin
  name: kubernetes-admin@kubernetes
  current-context: kubernetes-admin@kubernetes
  kind: Config
  preferences: {}
  users:
- name: kubernetes-admin
  user:
  client-certificate-data: DATA+OMITTED
  client-key-data: DATA+OMITTED

What is the current context set to in the `my-kube-config` file?

`kubectl config current-context --kubeconfig my-kube-config`
for default config: `kubectl config current-context`

I would like to use the `dev-user` to access `test-cluster-1`. Set the current context to the right one so I can do that.

Once the right context is identified, use the `kubectl config use-context` command.

To use that context, run the command: `kubectl config --kubeconfig=/root/my-kube-config use-context research`

To know the current context, run the command: `kubectl config --kubeconfig=/root/my-kube-config current-context`

We don't want to have to specify the kubeconfig file option on each command.

Set the `my-kube-config` file as the default kubeconfig by overwriting the content of `~/.kube/config` with the content of the `my-kube-config` file.

Replace the contents in the `default` kubeconfig file with the content from `my-kube-config` file with following command.

```
cp my-kube-config ~/.kube/config
```

identify the authorization modes configured on the cluster.

Check the `kube-apiserver` settings.

`kubectl describe pod kube-apiserver-controlplane -n kube-system` and look for `--authorization-mode`.

`kubectl get roles`

Which account is the `kube-proxy` role assigned to?

Run the command: `kubectl describe rolebinding kube-proxy -n kube-system`

if the user can list pods in the `default` namespace.

`kubectl get pods --as dev-user` and `kubectl auth can-i get pods --as dev-user`

For creating role:

`kubectl create role pod-reader --verb=get --verb=list --verb=watch --resource=pods`

For creating rolebinding:

`kubectl create rolebinding dev-user-binding --role=developer  --user=dev-user`

create sa

Run the command `kubectl create serviceaccount dashboard-sa`

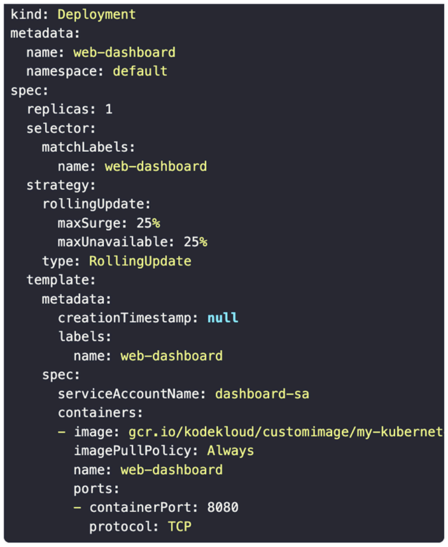
`kubectl create token dashboard-sa`

You shouldn't have to copy and paste the token each time. The Dashboard application is programmed to read token from the secret mount location. However currently, the `default` service account is mounted. Update the deployment to use the newly created ServiceAccount

Edit the deployment to change ServiceAccount from `default` to `dashboard-sa`.

Make use of the `kubectl set` command. Run the following command to use the newly created service account: - `kubectl set serviceaccount deploy/web-dashboard dashboard-sa`

we have to add serviceAccount section in the spec section of pod under deployment manifest



```
kind: Deployment
metadata:
  name: web-dashboard
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      name: web-dashboard
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        name: web-dashboard
    spec:
      serviceAccountName: dashboard-sa
      containers:
      - image: gcr.io/kodekloud/customimage/my-kubernet
        imagePullPolicy: Always
        name: web-dashboard
        ports:
        - containerPort: 8080
          protocol: TCP
```

what type of secret used for docker registry?

`kubectl create secret --help`
Create a secret with specified type.

A docker-registry type secret is for accessing a container registry.

A generic type secret indicate an Opaque secret type.

A tls type secret holds TLS certificate and its associated key.

now we are upadting the regisrty for the image nginx-alpine

We decided to use a modified version of the application from an internal private registry. Update the image of the deployment to use a new image from `myprivateregistry.com:5000`

`myprivateregistry.com:5000`/nginx-alpine

-

Create a secret object with the credentials required to access the registry.

Secret: private-reg-cred

Secret Type: docker-registry

Secret Data

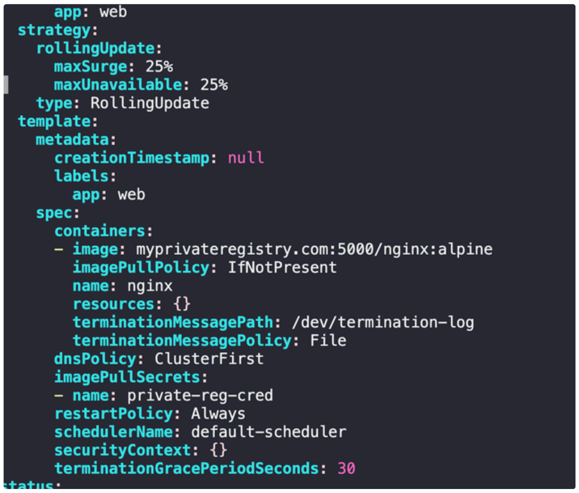
Name: `private-reg-cred`
Username: `dock_user`
Password: `dock_password`
Server: `myprivateregistry.com:5000`
Email: `dock_user@myprivateregistry.com`

Note: first we need to secret type and then name

Run the command: `kubectl create secret docker-registry private-reg-cred --docker-username=dock_user --docker-password=dock_password --docker-server=myprivateregistry.com:5000 --docker-email=dock_user@myprivateregistry.com`

Configure the deployment to use credentials from the new secret to pull images from the private registry

Edit deployment using `kubectl edit deploy web` command and add imagePullSecrets section. Use `private-reg-cred`.



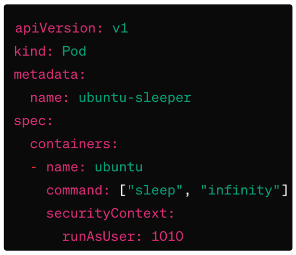
```
      app: web
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: web
    spec:
      containers:
      - image: myprivateregistry.com:5000/nginx:alpine
        imagePullPolicy: IfNotPresent
        name: nginx
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      imagePullSecrets:
      - name: private-reg-cred
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
status:
```

What is the user used to execute the sleep process within the `ubuntu-sleeper` pod?

Run the command: `kubectl exec ubuntu-sleeper -- whoami` and check the user that is running the container.

Edit the pod `ubuntu-sleeper` to run the sleep process with user ID `1010`.

Note: Only make the necessary changes. Do not modify the name or image of the pod.



```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper
spec:
  containers:
  - name: ubuntu
    command: ["sleep", "infinity"]
    securityContext:
      runAsUser: 1010
```

Update pod `ubuntu-sleeper` to run as Root user and with the `SYS_TIME` capability.

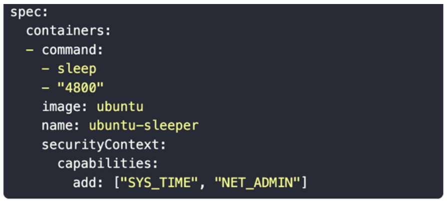
To delete the existing pod:

```
kubectl delete po ubuntu-sleeper
```

After that apply solution manifest file to add capabilities in `ubuntu-sleeper` pod:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper
  namespace: default
spec:
  containers:
  - command:
    - sleep
    - "4800"
    image: ubuntu
    name: ubuntu-sleeper
    securityContext:
      capabilities:
        add: ["SYS_TIME"]
```

then run the command `kubectl apply -f <file-name>.yaml` to create a pod from given definition file.



```
spec:
  containers:
  - command:
    - sleep
    - "4800"
    image: ubuntu
    name: ubuntu-sleeper
    securityContext:
      capabilities:
        add: ["SYS_TIME", "NET_ADMIN"]
```

Generic commands

The application stores logs at location `/log/app.log`.

`kubectl exec webapp -- cat /log/app.log`

To use that context, run the command: kubectl config --kubeconfig=/root/my-kube-config use-context research To know the current context, run the command: kubectl config --kubeconfig=/root/my-kube-config current-context

for yaml files creation
`kubectl run redis --image=redis123 --dry-run=client -oyaml >pod.yml`
for pod creation
`kubectl create -f pod.yml`

Run the command: You can check for apiVersion of replicaset by command `kubectl api-resources | grep replicaset`

`kubectl explain replicaset | grep VERSION` and correct the `apiVersion` for `ReplicaSet`.

Then run the command: `kubectl create -f /root/replicaset-definition-1.yaml`

To scale up to 5 PODs:

`  kubectl scale rs new-replica-set --replicas=5`

You can exec in to the container and open the file:
`kubectl exec webapp -- cat /log/app.log`

DaemonSet :

An easy way to create a DaemonSet is to first generate a YAML file for a Deployment with the command
`kubectl create deployment elasticsearch --image=registry.k8s.io/fluentd-elasticsearch:1.20 -n kube-system --dry-run=client -o yaml > fluentd.yaml.`

Next, remove the replicas, strategy and status fields from the YAML file using a text editor. Also, change the kind from Deployment to DaemonSet.

Finally, create the Daemonset by running `kubectl create -f fluentd.yaml`

If the POD was to get deleted now, would you be able to view these logs.
Use the command kubectl delete to delete a webapp pod and try to view those logs again.
The logs are stored in the Container's file system that lives only as long as the Container does. Once the pod is destroyed, you cannot view the logs again.

Scheduling

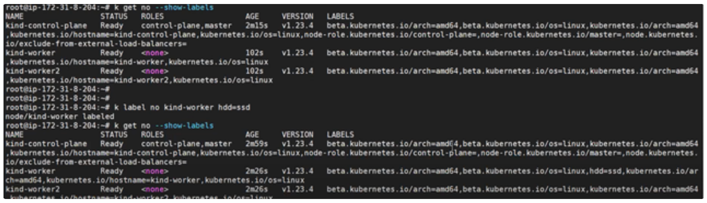
why ?

with default scheduling(if there is no schedule), scheduler will look for the node with good resource to schedule a pod.

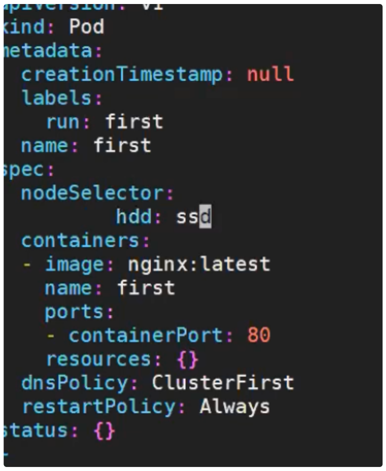
let's assume we have application it only run with ssd memory, and let's assume in cluster we don't have any node with ssd memory during that time with default scheduling schduler will select one of the good resource node but if that node has no ssd memory pod will be keeps on crashing.

types:

node selector: here we are labeling the node with key:value and in pod yaml section we are using nodeSelector attribute to select the labelled node.



```
root@ip-172-31-8-204:~# k get no --show-labels
NAME                 STATUS   ROLES                  AGE     VERSION   LABELS
kind-control-plane   Ready    control-plane,master   2m15s   v1.23.4   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64
,kubernetes.io/hostname=kind-control-plane,kubernetes.io/os=linux,node-role.kubernetes.io/control-plane=,node-role.kubernetes.io/master=,node.kubernetes.
io/exclude-from-external-load-balancers=
kind-worker          Ready    <none>                 102s    v1.23.4   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64
,kubernetes.io/hostname=kind-worker,kubernetes.io/os=linux
kind-worker2         Ready    <none>                 102s    v1.23.4   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64
,kubernetes.io/hostname=kind-worker2,kubernetes.io/os=linux
root@ip-172-31-8-204:~#
root@ip-172-31-8-204:~#
root@ip-172-31-8-204:~# k label no kind-worker hdd=ssd
node/kind-worker labeled
root@ip-172-31-8-204:~# k get no --show-labels
NAME                 STATUS   ROLES                  AGE     VERSION   LABELS
kind-control-plane   Ready    control-plane,master   2m59s   v1.23.4   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64
,kubernetes.io/hostname=kind-control-plane,kubernetes.io/os=linux,node-role.kubernetes.io/control-plane=,node-role.kubernetes.io/master=,node.kubernetes.
io/exclude-from-external-load-balancers=
kind-worker          Ready    <none>                 2m26s   v1.23.4   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,hdd=ssd,kubernetes.io/ar
ch=amd64,kubernetes.io/hostname=kind-worker,kubernetes.io/os=linux
kind-worker2         Ready    <none>                 2m26s   v1.23.4   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64
,kubernetes.io/hostname=kind-worker2,kubernetes.io/os=linux
```

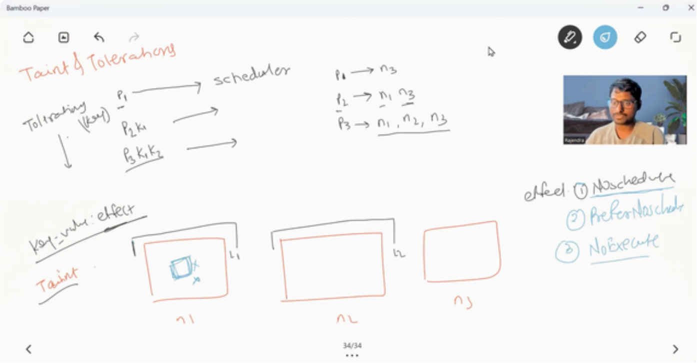


```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: first
  name: first
spec:
  nodeSelector:
          hdd: ssd
  containers:
  - image: nginx:latest
    name: first
    ports:
    - containerPort: 80
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

node affinity: it is same as node selector but we are using complex labelling on the pod

Taints and tolerations:

Taints mainly we are going to attach to the node and toleration will be writting in pod.

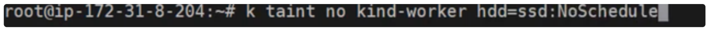


effects:

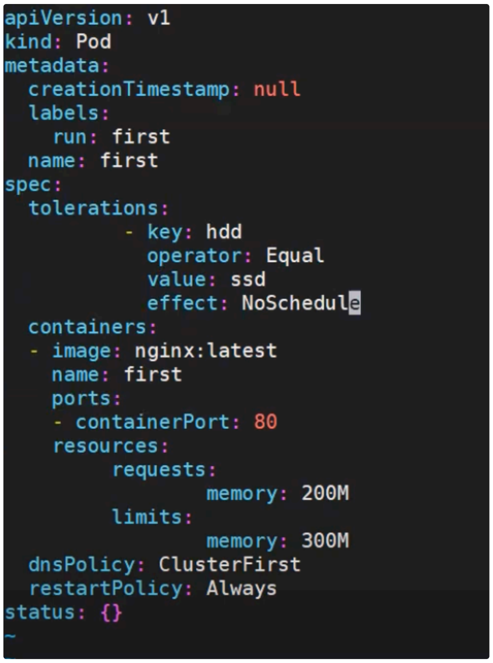
noscheduler: let’s assume already a pod is running on the node now am tainting the node with effect noscheduler. so with this effect it won’t effect running pod but it will not allow for the pod has no toleration.

Prefernoscheduler: in this effect earlier it will not allow for pod has no tolerations but if it not finds any good resource node finaaly it will allow

noexecute: let’s assume already pod is running on the node, now am tainting the node with effect noexecute so it will evict the pod from the node and also it will allow only toleraations pod.



```
root@ip-172-31-8-204:~# k taint no kind-worker hdd=ssd:NoSchedule
```



```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: first
  name: first
spec:
  tolerations:
          - key: hdd
            operator: Equal
            value: ssd
            effect: NoSchedule
  containers:
  - image: nginx:latest
    name: first
    ports:
    - containerPort: 80
    resources:
         requests:
                 memory: 200M
         limits:
                 memory: 300M
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
~
```

Commands:

```
kubectl label node worker-1 tier=frontend
```

```
kubectl get nodes --show-labels
```

Tainting node:

```
kubectl taint nodes controlplane node-role.kubernetes.io/control-plane:NoSchedule
```

Untaining node:

```
kubectl taint nodes controlplane node-role.kubernetes.io/control-plane:NoSchedule-
```

Use selectors to filter the output

```
kubectl get pods --selector env=dev --no-headers | wc -l
```