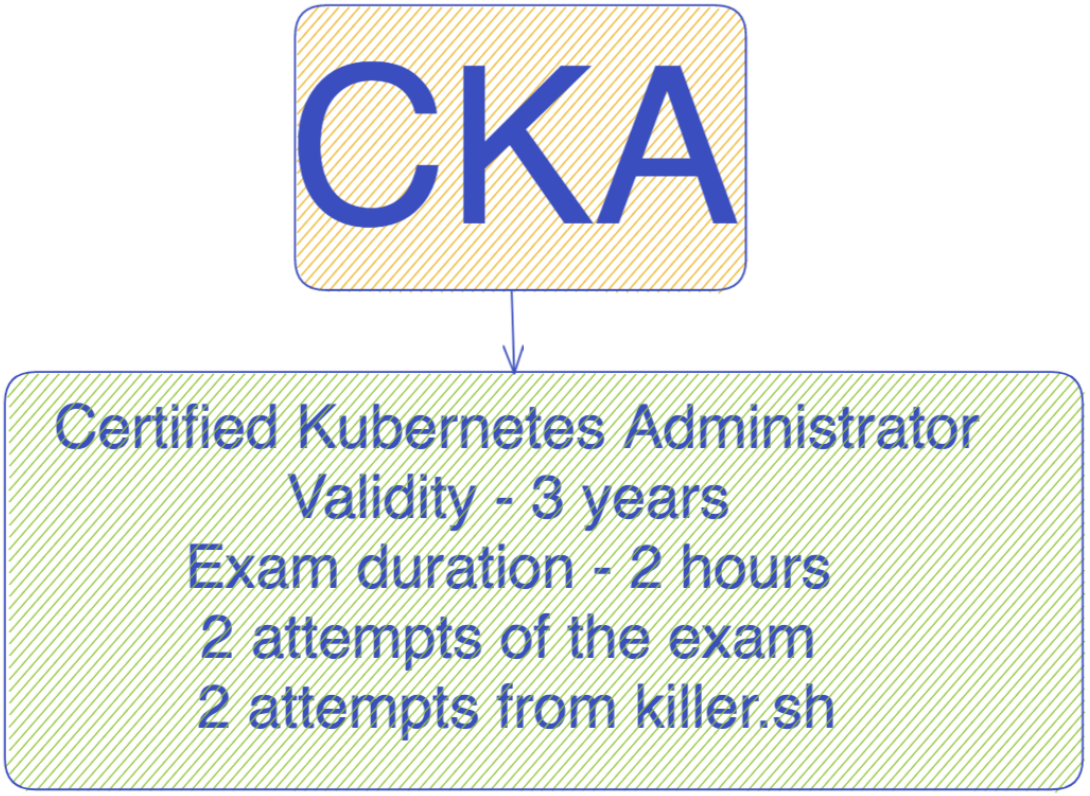
CKA - CERTIFIED KUBERNETES ADMINISTRATOR

Exam duration - 2 hours



This book covers many different scenarios, how to create them and the detailed solutions. Explanations and concepts have also been explained at various places.

My recommendation is to do all these scenarios along with the theory material which is covered in my four hour Kubernetes workshop available on my youtube channel.

Kubernetes 101 workshop - complete hands-on

Kubesimplify
1.1M views • Streamed 1 year ago

Link to the workshop

Acknowledgement

Finally, the book is coming out after countless hours of dedication and my love for Kubernetes. I would like to thank my friends for proof reading the book including Srinivasula Reddy Karnati, Kunal Verma and Aakash Nagpal.

While many have supported and guided me throughout this journey, there is one special acknowledgment I wish to make.

To my brother Aseem Pathak, I would like to dedicate this book to you, you have taught me everything and were my sole strength when all seemed lost.
You left us in 2021, our shared memories continue to inspire me. This book is a tribute to our bond, our shared dreams, and the countless moments we've cherished.

For everyone reading, I lost my brother to Covid but I will continue to shine his name wherever I can. He has taught me all the valuable life lessons.

Table of Contents

Scenario 1: Create a Kubernetes cluster using kubeadm.

Solution 1-

Knowing how to create a Kubernetes cluster is important in my opinion. You should be able to use Kubeadm and Ubuntu machines to create a 1.27 Kubernetes cluster.
Below is the script to create a 1.27 Kuberentes cluster using Ubuntu 22.04 instances. You do not need to memorise the commands to create the cluster. The basic command to initiate the process is kubeadm init, and the rest is the installation of CRI(containerd), kubelet, kubectl, and swap memory stuff.

For this scenario I am running this on an Ubuntu instance and you can choose any Ubuntu Instance and will be using the same setup for all the other scenarios.

Run the below commands:

```
echo "step1- install kubectl,kubeadm and kubelet 1.27.1"

curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo
apt-key add -
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee
/etc/apt/sources.list.d/kubernetes.list
echo "kubeadm install"
sudo apt update -y
sudo apt -y install vim git curl wget kubelet=1.27.1-00
kubeadm=1.27.1-00 kubectl=1.27.1-00

echo "memory swapoff"
sudo sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
sudo swapoff -a
sudo modprobe overlay
sudo modprobe br_netfilter

echo "Containerd setup"
sudo tee /etc/modules-load.d/containerd.conf <<EOF
overlay
br_netfilter
EOF
sudo tee /etc/sysctl.d/kubernetes.conf<<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
sysctl --system
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -
sudo add-apt-repository -y "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

```
sudo apt update -y
echo -ne '\n' | sudo apt-get -y install containerd
mkdir -p /etc/containerd
containerd config default > /etc/containerd/config.toml
sudo systemctl restart containerd
sudo systemctl enable containerd
sudo sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g'
/etc/containerd/config.toml
sudo systemctl restart containerd
sudo systemctl enable kubelet
echo "image pull and cluster setup"
sudo kubeadm config images pull --cri-socket
unix:///run/containerd/containerd.sock --kubernetes-version v1.27.1
sudo kubeadm init   --pod-network-cidr=10.244.0.0/16   --upload-certs
--kubernetes-version=v1.27.1  --control-plane-endpoint=$(hostname)
--ignore-preflight-errors=all  --cri-socket
unix:///run/containerd/containerd.sock
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
export KUBECONFIG=/etc/kubernetes/admin.conf
echo "Apply flannel network"
kubectl apply -f
https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel.
yml
kubectl taint node $(hostname)
node-role.kubernetes.io/control-plane:NoSchedule-
```

Once you run the above script by saving it in a sh file and running it or running each command separately, your 1.27.1 cluster will be ready in a few minutes.

```
kubectl get nodes
NAME                        STATUS   ROLES           AGE     VERSION
controlplane-a5cc-270721    Ready    control-plane   9m52s   v1.27.1
```

You can use this cluster throughout all the other scenarios.

Congratulations, you have set your first Kubernetes cluster from scratch on an Ubuntu 22.04 instance. It is a single-node cluster that is only for learning purposes. To make things production-ready, you need to have three control planes with an HAproxy or Kube-vip load balancing them, and then you can have any number of worker nodes connected.

Talking about highly available clusters, you can set up that using kubeadm as well. So, for example, you can have a separate etcd cluster that sits outside the Kubernetes cluster.

From the Docs:

- https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/

```
etcd:
  external:
    endpoints:
      - https://ETCD_0_IP:2379 # change ETCD_0_IP appropriately
      - https://ETCD_1_IP:2379 # change ETCD_1_IP appropriately
      - https://ETCD_2_IP:2379 # change ETCD_2_IP appropriately
```

When etcd is external, it is given like above in the ClusterConfiguration file.

- https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/ : This explains about various topology methods

Let's also add worker nodes to this cluster. So we can use the same cluster throughout all the scenarios. In order to do that you have to create two more instances and then set till this command from above script.

`sudo kubeadm config images pull --cri-socket unix:///run/containerd/containerd.sock --kubernetes-version v1.27.1`

Now once it's done till here. From the control plane node you need to copy the join command. Your join command will be separate from mine.

```
Then you can join any number of worker nodes by running the following on each as root:

kubeadm join controlplane-a5cc-270721:6443 --token g5srpp.i2l0oigoizixnwj9 \
    --discovery-token-ca-cert-hash
sha256:f2bea9ea63993f24358736658182cea53ce0550217668a42e2ef98c484cb390e
```

Run this command on both the worker nodes and you will see the nodes getting joined to the cluster. You might need to add the public Ip and the hostname of the controlplane to your `/etc/hosts` file to make it work.

```
kubeadm join controlplane-a5cc-270721:6443 --token g5srpp.i2l0oigoizixnwj9     --discovery-
token-ca-cert-hash sha256:f2bea9ea63993f24358736658182cea53ce0550217668a42e2ef98c484cb390e
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm
kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file
"/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

```
kubectl get nodes
NAME                      STATUS   ROLES           AGE     VERSION
controlplane-a5cc-270721  Ready    control-plane   166m    v1.27.1
worker-1-937f-617615      Ready    <none>          12s     v1.27.1
worker-2-e26c-617615      Ready    <none>          3m47s   v1.27.1
```

Scenario 2 - Create a Pod with name saiyam using image nginx

Solution 2-

For this particular scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

You can use an imperative command as shown below to directly run a single pod with a given name and image name, and you can also check the image name using -o jsonpath. Since no namespace is specified, I have not specified the namespace so the pod should get created in the default namespace.

```
kubectl run saiyam --image=nginx
kubectl get pods  -o jsonpath="{.items[*].spec.containers[*].image}" |tr
-s '[[:space:]]' '\n' |sort |uniq -c
```

```
demo ~ kubectl run saiyam --image=nginx
pod/saiyam created
demo ~ kubectl get pods
NAME     READY   STATUS    RESTARTS   AGE
saiyam   1/1     Running   0          4s
demo ~ kubectl get pods  -o jsonpath="{.items[*].spec.containers[*].image}" |tr -s
'[[:space:]]' '\n' |sort |uniq -c
      1 nginx
```

-o jsonpath will also be used in other scenarios, so it's good to get comfortable with this output command.

A pod is said to be the smallest unit of your node, it consists of one or more containers that will be your application and some helper/sidecar containers. Usually, in most cases it's one container per pod. Each pod gets its unique ip address as well and whenever the pod dies the ip will get changed. You will end up using higher level abstractions like DaemonSet or Deployment rather than the pod directly but it's important to know the fundamentals.

Cleanup

```
Kubectl delete pods --all
```

Scenario 3 - Init container

Create a Pod with name saiyam and image nginx and later add init container with name sam-init and image busybox. Make the init container update /usr/share/nginx/html with “hello world”.

Solution 3 -

Init containers are the containers that are executed before the main container and run to completion. It can be used to set up stuff for the main container, it can change the filesystem before the main app container starts if needed and it can also reduce the attack surface by keeping some checks as part of init containers. There can be multiple init containers and all will run sequentially, as defined in the YAML spec. The probes(liveness and readiness etc) cannot be used for init containers.
First, we get the `--dry-run=client -oyaml` for the pod saiyam and then add the init container to do the magic. `--dry-run=client` command is used to generate a base YAML file that can be helpful during the exam and save time.

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

Getting the YAML file

```
kubectl run saiyam --image=nginx --dry-run=client -oyaml > pod.yaml
```

```
demo ~ kubectl run saiyam --image=nginx --dry-run=client -oyaml > pod.yaml
demo ~ cat pod.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: saiyam
  name: saiyam
spec:
  containers:
  - image: nginx
    name: saiyam
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Adding init container and volume mount to the nginx container.

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: saiyam
  name: saiyam
spec:
  containers:
  - image: nginx
    name: saiyam
    resources: {}
    volumeMounts:
    - name: demo
      mountPath: /usr/share/nginx/html
  initContainers:
  - name: sam-init
    image: busybox:1.28
    command: ['sh', '-c', 'echo "hello world" | tee /tmp/index.html']
    volumeMounts:
    - name: demo
      mountPath: "/tmp/"
  dnsPolicy: Default
  volumes:
  - name: demo
    emptyDir: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Apply the pod

```
demo ~ kubectl apply -f pod.yaml
pod/saiyam created
demo ~ kubectl get pods
NAME     READY   STATUS            RESTARTS   AGE
saiyam   0/1     Init:0/1   0          6s
demo ~ kubectl get pods
NAME     READY   STATUS            RESTARTS   AGE
saiyam   0/1     PodInitializing   0          6s
demo ~ kubectl get pods
NAME     READY   STATUS    RESTARTS   AGE
saiyam   1/1     Running   0          7s
```

Exec into the pod and curl on localhost; you will see the new file created by the init container, and the same file got mounted inside the nginx container.

```
kubectl exec -it saiyam -- sh
```

```
demo ~ kubectl exec -it saiyam -- sh

Defaulted container "saiyam" out of: saiyam, sam-init (init)
# curl localhost
hello world
```

Cleanup:

```
kubectl delete -f pod.yaml
pod "saiyam" deleted
```

Scenario 4 - Multi container Pod

Create a multi-container pod with one container with name=**demo** and image=**nginx**, another container with name=**database** with image=**redis**.

Solution 4 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

First, use the --dry-run command to get the yaml file

```
kubectl run demo --image=nginx --dry-run=client -oyaml > pod.yaml
```

Edit the pod.yaml file and add another container

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: demo
  name: demo
spec:
  containers:
  - image: nginx
    name: demo
    resources: {}
  - image: redis
    name: database
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Apply the file

```
demo ~ kubectl apply -f pod.yaml
pod/demo created
demo ~ kubectl get pods
NAME   READY   STATUS              RESTARTS   AGE
demo   0/2     ContainerCreating   0          2s
demo ~ kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
demo   2/2     Running   0          6s
```

As you can see, 2/2 ready means there are two containers in this pod running. Multi-container pods can have different use cases, with one of the most common ones being sidecar patterns that uses an envoy proxy attached as a sidecar. Another use case can be for logging, reverse proxies to serve files from a website via nginx.

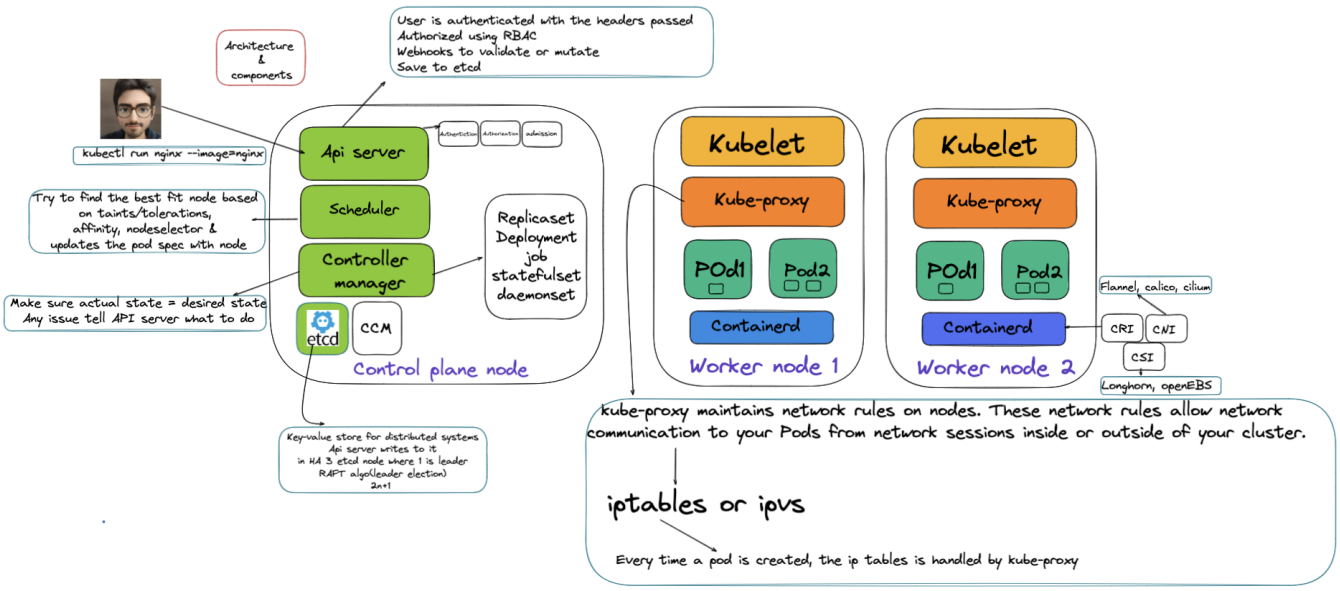
Another interesting thing to note here is that when the containers run in the same pod, they share the same network namespace and can communicate with each other over localhost.

```
kubectl get pods -owide
NAME    READY    STATUS     RESTARTS    AGE    IP            NODE
NOMINATED NODE   READINESS GATES
demo    2/2      Running    0           82s    10.244.2.7    worker-1-937f-617615
<none>           <none>
```

You can see that the IP address is the same for both containers.

Now you can be asked to create multiple containers in a single pod; you need to get the skeleton file from the `--dry-run` command and keep adding the containers. Ensure the container's name and image are as per the question.

Below is what the request to create the pod looks like with different Kubernetes components in action.



Cleanup

```
kubectl delete -f pod.yaml
pod "demo" deleted
```

Scenario 5 - Labels and Selectors

Assign a label to the node **CKA=true** and then create a pod(**name=demo,image=nginx**) and assign it to the node with the label you created.

Solution 5 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

Labels and selectors are handy for services to know which pods to send the traffic to.

Now when you talk about Nodes, you can give labels to the node and mark them in a way that you can schedule workloads of different types (eg. Memory intensive or GPU workloads) to specific nodes.

Label the node (you can label any node here)

```
kubectl get nodes
NAME                          STATUS   ROLES           AGE     VERSION
controlplane-a5cc-270721      Ready    control-plane   3h18m   v1.27.1
worker-1-937f-617615          Ready    <none>          31m     v1.27.1
worker-2-e26c-617615          Ready    <none>          35m     v1.27.1
```

```
kubectl label node worker-1-937f-617615  CKA=true
node/worker-1-937f-617615 labeled
```

```
demo ~ kubectl get nodes
NAME                      STATUS   ROLES           AGE     VERSION
controlplane-a5cc-270721   Ready    control-plane   3h19m   v1.27.1
worker-1-937f-617615       Ready    <none>          33m     v1.27.1
worker-2-e26c-617615       Ready    <none>          36m     v1.27.1
demo ~ kubectl get node worker-1-937f-617615 --show-labels
NAME                     STATUS   ROLES    AGE   VERSION   LABELS
worker-1-937f-617615   Ready    <none>   33m   v1.27.1
CKA=true,beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,k
ubernetes.io/hostname=worker-1-937f-617615,kubernetes.io/os=linux
demo ~ kubectl get node worker-1-937f-617615 --show-labels | grep CKA
worker-1-937f-617615   Ready    <none>   33m   v1.27.1
CKA=true,beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,k
ubernetes.io/hostname=worker-1-937f-617615,kubernetes.io/os=linux
```

Create a pod manifest using `--dry-run` and then use with **nodeSelector**.

```
kubectl run demo --image=nginx --dry-run=client -oyaml > pod.yaml
```

Add below to the pod spec section:

```
nodeSelector:
    CKA: "true"
```

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: demo
  name: demo
spec:
  nodeSelector:
    CKA: "true"
  containers:
  - image: nginx
    name: demo
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Apply the manifest

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

```
demo ~ kubectl apply -f pod.yaml
pod/demo created
demo ~ kubectl get pods -owide
NAME   READY   STATUS    RESTARTS   AGE   IP           NODE                    NOMINATED NODE
  READINESS GATES
demo   1/1     Running   0          18s   10.244.2.8   worker-1-937f-617615   <none>
  <none>
```

You can see that the pod got created and started running on the worker-1-937f-617615 node, and this is the node where we added the label.

Cleanup

```
kubectl delete -f pod.yaml
pod "demo" deleted
```

Scenario 6 - Assign a pod (name demo, image=nginx) to a particular node.

Solution 6 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

In the above scenario (Scenario 5), we saw how to label a node and then assign a pod using `nodeSelector`. However, if you know the node name where you want the workload to be scheduled; in that case, you can use `nodeName` directly to assign a pod to the node.

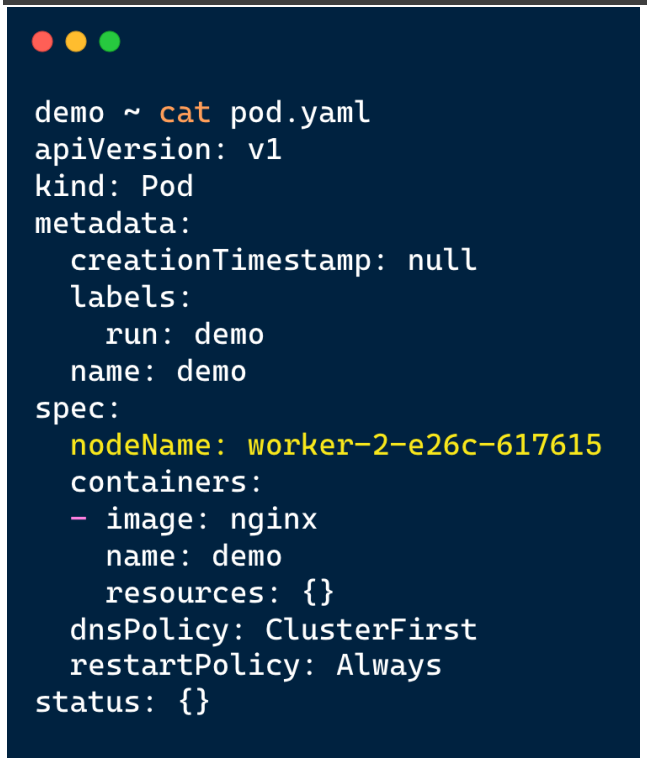
Create a pod manifest using `--dry-run` and then add **nodeName** to the spec section.

```
kubectl run demo --image=nginx --dry-run=client -oyaml > pod.yaml
```

Add this to the pod spec section. Since we have three nodes, so let's schedule this pod to `worker-2-e26c-617615`

```
kubectl get nodes
NAME                      STATUS   ROLES           AGE     VERSION
controlplane-a5cc-270721  Ready    control-plane   3h31m   v1.27.1
worker-1-937f-617615      Ready    <none>          45m     v1.27.1
worker-2-e26c-617615      Ready    <none>          48m     v1.27.1
```

```
nodeName: worker-2-e26c-617615
```



Apply the pod.yaml file

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

```
kubectl get pods -owide
NAME   READY   STATUS    RESTARTS   AGE   IP           NODE                   NOMINATED NODE
  READINESS GATES
demo   1/1     Running   0          56s   10.244.1.3   worker-2-e26c-617615   <none>
  <none>
```

Above method demonstrates another way using which you can assign a pod to a node where you know the name; in this case, we assigned it to worker-2-e26c-617615 node and can see that the pod started creating on the node specified.

Cleanup

```
kubectl delete -f pod.yaml
pod "demo" deleted
```

Scenario 7 - Static pods

Create a static pod on one of the worker nodes with the name: demo and image nginx.

Solution 7 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

Static pods are managed by kubelet and not by the api-server; you can see the kubelet configuration to know where the `staticPodPath` points to and create the pod manifest in that directory.

```
cat /var/lib/kubelet/config.yaml | grep static
staticPodPath: /etc/kubernetes/manifests
```

We have a three node cluster:

```
kubectl get nodes
NAME                          STATUS   ROLES           AGE     VERSION
controlplane-a5cc-270721      Ready    control-plane   3h38m   v1.27.1
worker-1-937f-617615          Ready    <none>          52m     v1.27.1
worker-2-e26c-617615          Ready    <none>          56m     v1.27.1
```

Generate the yaml for the pod to be created:

```
kubectl run demo --image=nginx --dry-run=client -oyaml
```

Copy the manifest file and ssh into worker1 node.

```
root@controlplane-a5cc-270721:~# kubectl run demo --image=nginx --dry-run=client -oyaml

apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: demo
  name: demo
spec:
  containers:
  - image: nginx
    name: demo
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Check staticPodPath

```
cat /var/lib/kubelet/config.yaml | grep static
```

```
root@worker-1-937f-617615:~# cat /var/lib/kubelet/config.yaml | grep static
staticPodPath: /etc/kubernetes/manifests
```

Now copy the pod manifest you created in the above step on the controlplane node and put that as pod.yaml in the `/etc/kubernetes/manifests` folder.

```
worker1 ~ cd /etc/kubernetes/manifests/
worker1 ~ vi pod.yaml
worker1 ~ cat pod.yaml

apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: demo
  name: demo
spec:
  containers:
  - image: nginx
    name: demo
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Now go back to the controlplane node and see if the pod gets created.

```
controlplane ~ kubectl get pods -w
NAME                          READY   STATUS    RESTARTS   AGE
demo-worker-1-937f-617615   1/1     Pending   0          17s
NAME                          READY   STATUS    RESTARTS   AGE
demo-worker-1-937f-617615   1/1     Running   0          19s
```

As soon as you add the pod in the manifest directory of the node, you will see that the pod starts running, and this pod is managed by kubelet present on worker-1.
So the pod you see as the output of kubectl get pods is called a **mirror pod**. The Pod names will be suffixed with the node hostname with a leading hyphen.

Sometimes there can be a misconfiguration in the path of staticPodPath that can lead to cluster break, and you need to check if the path is right for static pods.

Cleanup
Remove the pod.yaml file from worker1 `/etc/kubernetes/manifests` directory.

```
rm /etc/kubernetes/manifests/pod.yaml
```

Scenario 8 - Highest memory consumption

Get the highest memory consuming pod and put that in a file `/tmp/high-mem.txt`

Solution 8 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

In order to get prepared for the scenario you need to install the metrics server. If you are using K3s, then it will come pre-bundled with it.

Install metrics server

```
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/downlo
ad/components.yaml
```

```
controlplane ~ kubectl apply -f https://github.com/kubernetes-sigs/metrics-
server/releases/latest/download/components.yaml
serviceaccount/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
service/metrics-server created
deployment.apps/metrics-server created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
```

The probe might fail due to certificate issue, so you might need to add `--kubelet-insecure-tls` flag to the metrics server deployment.

```
kubectl edit deploy metrics-server -n kube-system
```

```
- args:
        - --cert-dir=/tmp
        - --secure-port=4443
        - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
        - --kubelet-use-node-status-port
        - --metric-resolution=15s
        - --kubelet-insecure-tls
```

```
kubectl edit deploy metrics-server -n kube-system
deployment.apps/metrics-server edited
```

Now you will be able to see the metrics server running.

```
controlplane ~ kubectl get pods -A | grep metrics
kube-system    metrics-server-75f45b4dd4-95x5f                  1/1     Running   0
70s
```

You can now easily get the CPU or memory info for the pods once the metrics server is installed.

```
kubectl top pods --all-namespaces  --sort-by memory
```

```
controlplane ~ kubectl top pods --all-namespaces  --sort-by memory
NAMESPACE      NAME                                              CPU(cores)   MEMORY(bytes)
kube-system    kube-apiserver-controlplane-a5cc-270721           60m          265Mi
kube-system    kube-controller-manager-controlplane-a5cc-270721  26m          43Mi
kube-system    etcd-controlplane-a5cc-270721                     29m          43Mi
kube-system    kube-scheduler-controlplane-a5cc-270721           6m           18Mi
kube-system    metrics-server-75f45b4dd4-95x5f                   7m           16Mi
kube-system    coredns-5d78c9869d-8kt68                          5m           12Mi
kube-system    coredns-5d78c9869d-8dqbb                          2m           12Mi
kube-system    kube-proxy-zjxvb                                  1m           11Mi
kube-flannel   kube-flannel-ds-8vzlc                             14m          11Mi
kube-flannel   kube-flannel-ds-bkbmv                             16m          11Mi
kube-system    kube-proxy-fmb6r                                  1m           11Mi
kube-system    kube-proxy-h5vvx                                  2m           11Mi
kube-flannel   kube-flannel-ds-zgl5g                             13m          10Mi
```

You can also sort by cpu using `--sort-by cpu`

Now, you need to get the name of the highest memory used pod and put that in a file. There are various command line hacks to do that, and one of the ways is:

```
kubectl top pods --all-namespaces  --sort-by memory  --no-headers=true
|awk '{print $2}' | head -n 1
kube-apiserver-controlplane-a5cc-270721
```

Put that into a file

```
kubectl top pods --all-namespaces  --sort-by memory  --no-headers=true
|awk '{print $2}' | head -n 1 > /tmp/high-mem.txt
```

```
controlplane ~ kubectl top pods --all-namespaces  --sort-by memory  --no-headers=true |awk
'{print $2}' | head -n 1 > /tmp/high-mem.txt
controlplane ~ cat /tmp/high-mem.txt
kube-apiserver-controlplane-a5cc-270721
```

Now you can play around doing for cpu/memory in different namespaces and put the output into a file.

Cleanup
You can remove the metrics server simply by deleting the manifest that was applied.

```
kubectl delete -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/downlo
ad/components.yaml
```

Scenario 9 - Scheduling without nodeName and NodeSelector

Create two pods, first with name=nginx and image=nginx, second with name=demo and image=redis. Make sure that they end up on the same node. You cannot use nodeName or nodeselector.

Solution 9 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

Here we will use a new concept called pod affinity. So when you define the pods, you can define their affinities for the nodes and pods.

```
nodeAffinity  <NodeAffinity>
  Describes node affinity scheduling rules for the pod.

podAffinity   <PodAffinity>
  Describes pod affinity scheduling rules (e.g. co-locate this pod in the same
  node, zone, etc. as some other pod(s)).

podAntiAffinity   <PodAntiAffinity>
  Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in
  the same node, zone, etc. as some other pod(s)).
```

For this scenario, we have to use podAffinity and not nodeAffinity. PodAffinity is about scheduling pods based on the presence of other pods, while NodeAffinity is about scheduling pods based on node characteristics.

Create two pod yaml files using the --dry-run command

```
kubectl run nginx --image=nginx --dry-run=client -oyaml > pod1.yaml
kubectl run demo --image=redis --dry-run=client -oyaml > pod2.yaml
```

Add below to the pod spec section of pod2.yaml

```
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: demo
            operator: In
            values:
            - cka
        topologyKey: kubernetes.io/hostname
```

Add Label `demo: cka` to pod1.yaml

Here are two manifests

```
controlplane ~ cat pod1.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
    demo: cka
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

```
controlplane ~ cat pod2.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: demo
  name: demo
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: demo
            operator: In
            values:
            - cka
        topologyKey: kubernetes.io/hostname
  containers:
  - image: redis
    name: demo
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

```
kubectl apply -f .
pod/nginx created
pod/demo created
```

```
kubectl get pods -owide
NAME    READY   STATUS    RESTARTS   AGE   IP            NODE                    NOMINATED
NODE   READINESS GATES
demo    1/1     Running   0          8s    10.244.1.10   worker-2-e26c-617615   <none>
    <none>
nginx   1/1     Running   0          8s    10.244.1.9    worker-2-e26c-617615   <none>
    <none>
```

When you apply them, they will be scheduled on the same node. You can check on a kind cluster or k3d cluster with more nodes.

I have a 6-node cluster and will apply it multiple times to show that they will be scheduled and run on the same node.

```
demo ~ kubectl get nodes
NAME                                              STATUS   ROLES    AGE
k3s-demo2-db61-713665-node-pool-6a60-tmp4g   Ready    <none>   22m
k3s-demo2-db61-713665-node-pool-6a60-odnlg   Ready    <none>   22m
k3s-demo2-db61-713665-node-pool-6a60-oao3o   Ready    <none>   22m
k3s-demo2-db61-713665-node-pool-6a60-ne7dk   Ready    <none>   22m
k3s-demo2-db61-713665-node-pool-6a60-rwdxt   Ready    <none>   22m
k3s-demo2-db61-713665-node-pool-6a60-w1tdi   Ready    <none>   22m
```

```
demo ~ kubectl apply -f .
pod/nginx created
pod/demo created
demo ~ kubectl get pods -owide
NAME    READY   STATUS    RESTARTS   AGE   IP          NODE                                              NOMINATED NODE   READINES
S GATES
nginx   1/1     Running   0          6s    10.42.1.5   k3s-demo2-db61-713665-node-pool-6a60-ne7dk   <none>           <none>
demo    1/1     Running   0          6s    10.42.1.6   k3s-demo2-db61-713665-node-pool-6a60-ne7dk   <none>           <none>
```

Now let's cordon this node(by cordoning a node, you mark it as unavailable to the scheduler. Therefore, the node will not be eligible to host any new pods that will be added to your cluster.)

```
demo ~ kubectl cordon k3s-demo2-db61-713665-node-pool-6a60-ne7dk
node/k3s-demo2-db61-713665-node-pool-6a60-ne7dk cordoned
demo ~ kubectl delete -f .
pod "nginx" deleted
pod "demo" deleted
demo ~ kubectl apply -f .
pod/nginx created
pod/demo created
demo ~ kubectl get pods -owide
NAME    READY   STATUS    RESTARTS   AGE   IP          NODE                                              NOMINATED NODE   READINES
S GATES
nginx   1/1     Running   0          6s    10.42.2.6   k3s-demo2-db61-713665-node-pool-6a60-rwdxt   <none>           <none>
demo    1/1     Running   0          5s    10.42.2.7   k3s-demo2-db61-713665-node-pool-6a60-rwdxt   <none>           <none>
```

You can see that the pods are again scheduled on the same node!
This is how you can use podAffinity for this particular scenario. There are other affinities that you can study as well. Now try to use AntiAffinity to schedule them otherwise.

Cleanup

```
kubectl delete -f .
pod "nginx" deleted
pod "demo" deleted
```

Scenario 10 - Kubernetes Upgrade

Upgrade the kubernetes version 1.27.1 to 1.27.2 for control plane and worker-1

Solution 10 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

```
kubectl get nodes
NAME                      STATUS   ROLES           AGE   VERSION
controlplane-a5cc-270721  Ready    control-plane   11h   v1.27.1
worker-1-937f-617615      Ready    <none>          8h    v1.27.1
worker-2-e26c-617615      Ready    <none>          8h    v1.27.1
```

First, ssh onto the worker node and upgrade kubeadm to 1.27.2

```
apt -y install kubeadm=1.27.2-00
```

```
root@controlplane-a5cc-270721:~# apt -y install kubeadm=1.27.2-00
Reading package lists ... Done
Building dependency tree ... Done
Reading state information ... Done
The following packages will be upgraded:
  kubeadm
1 upgraded, 0 newly installed, 0 to remove and 205 not upgraded.
Need to get 9920 kB of archives.
After this operation, 4096 B of additional disk space will be used.
Get:1 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 kubeadm amd64
1.27.2-00 [9920 kB]
Fetched 9920 kB in 4s (2583 kB/s)
(Reading database ... 45674 files and directories currently installed.)
Preparing to unpack .../kubeadm_1.27.2-00_amd64.deb ...
Unpacking kubeadm (1.27.2-00) over (1.27.1-00) ...
Setting up kubeadm (1.27.2-00) ...
```

Plan the upgrade

```
kubeadm upgrade plan
```

```
root@controlplane-a5cc-270721:~# kubeadm upgrade plan 1.27.2
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm
kubeadm-config -o yaml'
[preflight] Running pre-flight checks.
[upgrade] Running cluster health checks
[upgrade] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: v1.27.1
[upgrade/versions] kubeadm version: v1.27.2
[upgrade/versions] Target version: 1.27.2
[upgrade/versions] Latest version in the v1.27 series: 1.27.2
W0804 00:23:29.356484  186836 compute.go:307] [upgrade/versions] could not find officially
supported version of etcd for Kubernetes 1.27.2, falling back to the nearest etcd version
(3.5.7-0)

Components that must be upgraded manually after you have upgraded the control plane with
'kubeadm upgrade apply':
COMPONENT   CURRENT       TARGET
kubelet     3 x v1.27.1   1.27.2

Upgrade to the latest version in the v1.27 series:

COMPONENT                 CURRENT   TARGET
kube-apiserver            v1.27.1   1.27.2
kube-controller-manager   v1.27.1   1.27.2
kube-scheduler            v1.27.1   1.27.2
kube-proxy                v1.27.1   1.27.2
CoreDNS                   v1.10.1   v1.10.1
etcd                      3.5.7-0   3.5.7-0

You can now apply the upgrade by executing the following command:

    kubeadm upgrade apply 1.27.2
```

The above will give you the information if anything needs to be manually upgraded.
Upgrade via kubeadm using `kubeadm upgrade apply 1.27.2`

```
root@controlplane-a5cc-270721:~# kubeadm upgrade apply 1.27.2
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm
kubeadm-config -o yaml'
[preflight] Running pre-flight checks.
[upgrade] Running cluster health checks
[upgrade/version] You have chosen to change the cluster version to "v1.27.2"
[upgrade/versions] Cluster version: v1.27.1
[upgrade/versions] kubeadm version: v1.27.2
[upgrade] Are you sure you want to proceed? [y/N]: y
```

Once you press enter, it will start downloading the new mages, renew the certificates etc.
After a little while you will see the cluster upgraded successfully.

```
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.27.2".
Enjoy!
```

```
[upgrade/kubelet] Now that your control plane is upgraded, please
proceed with upgrading your kubelets if you haven't already done so.
```

```
root@controlplane-a5cc-270721:~# kubectl get nodes
NAME                          STATUS   ROLES           AGE   VERSION
controlplane-a5cc-270721      Ready    control-plane   11h   v1.27.1
worker-1-937f-617615          Ready    <none>          8h    v1.27.1
worker-2-e26c-617615          Ready    <none>          8h    v1.27.1
```

Now cordon and drain the controlplane node for upgrading the kubelet and kubectl (you can directly do the drain as it will automatically cordon as well)

```
kubectl drain controlplane-a5cc-270721 --ignore-daemonsets
```

```
root@controlplane-a5cc-270721:~# kubectl drain controlplane-a5cc-270721 --ignore-daemonsets
node/controlplane-a5cc-270721 cordoned
Warning: ignoring DaemonSet-managed Pods: kube-flannel/kube-flannel-ds-zgl5g, kube-
system/kube-proxy-m8jrn
evicting pod kube-system/coredns-5d78c9869d-8dqbb
pod/coredns-5d78c9869d-8dqbb evicted
node/controlplane-a5cc-270721 drained
```

Update kubelet and kubectl:

```
apt-get install kubelet=1.27.2-00 kubectl=1.27.2-00
```

```
apt-get install kubelet=1.27.2-00 kubectl=1.27.2-00

Reading package lists ... Done
Building dependency tree ... Done
Reading state information ... Done
The following packages will be upgraded:
  kubectl kubelet
2 upgraded, 0 newly installed, 0 to remove and 203 not upgraded.
Need to get 29.0 MB of archives.
After this operation, 12.3 kB of additional disk space will be used.
Get:1 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 kubectl amd64
1.27.2-00 [10.2 MB]
Get:2 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 kubelet amd64
1.27.2-00 [18.7 MB]
Fetched 29.0 MB in 1s (25.5 MB/s)
(Reading database ... 45670 files and directories currently installed.)
Preparing to unpack .../kubectl_1.27.2-00_amd64.deb ...
Unpacking kubectl (1.27.2-00) over (1.27.1-00) ...
Preparing to unpack .../kubelet_1.27.2-00_amd64.deb ...
Unpacking kubelet (1.27.2-00) over (1.27.1-00) ...
Setting up kubectl (1.27.2-00) ...
Setting up kubelet (1.27.2-00) ...
```

Restart Kubelet and uncordon node

```
root@controlplane-a5cc-270721:~# sudo systemctl daemon-reload
root@controlplane-a5cc-270721:~# sudo systemctl restart kubelet
root@controlplane-a5cc-270721:~# kubectl uncordon controlplane-a5cc-270721
node/controlplane-a5cc-270721 uncordoned
```

```
root@controlplane-a5cc-270721:~# kubectl get nodes
NAME                          STATUS   ROLES           AGE   VERSION
controlplane-a5cc-270721      Ready    control-plane   11h   v1.27.2
worker-1-937f-617615          Ready    <none>          8h    v1.27.1
worker-2-e26c-617615          Ready    <none>          8h    v1.27.1
```

Now you can upgrade the worker nodes(by using any existing clusters) using the same approach:
- Cordon+Drain from the control plane node
- Update Kubeadm on worker1
- Kubeadm upgrade command `kubeadm upgrade node`
- Update kubectl and kubelet
- Restart Kubelet
- Uncordon Node

```
apt -y install kubeadm=1.27.2-00
sudo kubeadm upgrade node
apt-mark unhold kubelet kubectl && apt-get update && apt-get install -y kubelet=1.27.2-00
kubectl=1.27.2-00 && apt-mark hold kubelet kubectl
sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

```
root@controlplane-a5cc-270721:~# kubectl get nodes
NAME                          STATUS                     ROLES           AGE   VERSION
controlplane-a5cc-270721      Ready                      control-plane   11h   v1.27.2
worker-1-937f-617615          Ready,SchedulingDisabled   <none>          9h    v1.27.2
worker-2-e26c-617615          Ready                      <none>          9h    v1.27.1
root@controlplane-a5cc-270721:~# kubectl uncordon worker-1-937f-617615
node/worker-1-937f-617615 uncordoned
root@controlplane-a5cc-270721:~# kubectl get nodes
NAME                          STATUS   ROLES           AGE   VERSION
controlplane-a5cc-270721      Ready    control-plane   11h   v1.27.2
worker-1-937f-617615          Ready    <none>          9h    v1.27.2
worker-2-e26c-617615          Ready    <none>          9h    v1.27.1
```

Scenario 11 - etcd backup and restore

Take etcd backup and then add the backup to `/tmp/backup.db` location. Restore it to a different location and then point etcd to use that.

Solution 11-

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

First, check the etcd pod is running

```
kubectl get pods -n kube-system | grep etcd
etcd-controlplane-a5cc-270721                          1/1     Running   0
11h
```

Now describe the pod in order to get the certificate's details.

```
--advertise-client-urls=https://192.168.1.2:2379
      --cert-file=/etc/kubernetes/pki/etcd/server.crt
      --client-cert-auth=true
      --data-dir=/var/lib/etcd
      --experimental-initial-corrupt-check=true
      --experimental-watch-progress-notify-interval=5s
      --initial-advertise-peer-urls=https://192.168.1.2:2380
      --initial-cluster=controlplane-a5cc-270721=https://192.168.1.2:2380
      --key-file=/etc/kubernetes/pki/etcd/server.key
      --listen-client-urls=https://127.0.0.1:2379,https://192.168.1.2:2379
      --listen-metrics-urls=http://127.0.0.1:2381
      --listen-peer-urls=https://192.168.1.2:2380
      --name=controlplane-a5cc-270721
      --peer-cert-file=/etc/kubernetes/pki/etcd/peer.crt
      --peer-client-cert-auth=true
      --peer-key-file=/etc/kubernetes/pki/etcd/peer.key
      --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
      --snapshot-count=10000
      --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
```

Here you can see the `--trusted-ca-file`, `--cert-file` and the `--key-file`.

Now, use etcdctl to interact with etcd, it's a simple command line utility to interact with etcd.

Etcdctl install

```
wget
https://github.com/etcd-io/etcd/releases/download/v3.5.9/etcd-v3.5.9-lin
ux-amd64.tar.gz
tar -xvf etcd-v3.5.9-linux-amd64.tar.gz
mv etcd-v3.5.9-linux-amd64/etcdctl /usr/local/bin/
mv etcd-v3.5.9-linux-amd64/etcdutl /usr/local/bin/
```

```
sudo ETCDCTL_API=3 etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt
--cert=/etc/kubernetes/pki/etcd/server.crt
--key=/etc/kubernetes/pki/etcd/server.key snapshot save /tmp/backup.db
```

Here the --cacert = `--trusted-ca-file` --cert = `--cert-file` and key = `--key-file`

```
root@controlplane-a5cc-270721:~# sudo ETCDCTL_API=3 etcdctl --
cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --
key=/etc/kubernetes/pki/etcd/server.key snapshot save /tmp/backup.db
{"level":"info","ts":"2023-08-
04T00:58:39.463346Z","caller":"snapshot/v3_snapshot.go:65","msg":"created temporary db
file","path":"/tmp/backup.db.part"}
{"level":"info","ts":"2023-08-
04T00:58:39.471134Z","logger":"client","caller":"v3@v3.5.9/maintenance.go:212","msg":"opened
snapshot stream; downloading"}
{"level":"info","ts":"2023-08-
04T00:58:39.471823Z","caller":"snapshot/v3_snapshot.go:73","msg":"fetching
snapshot","endpoint":"127.0.0.1:2379"}
{"level":"info","ts":"2023-08-
04T00:58:39.515474Z","logger":"client","caller":"v3@v3.5.9/maintenance.go:220","msg":"complet
ed snapshot read; closing"}
{"level":"info","ts":"2023-08-
04T00:58:39.530346Z","caller":"snapshot/v3_snapshot.go:88","msg":"fetched
snapshot","endpoint":"127.0.0.1:2379","size":"3.4 MB","took":"now"}
{"level":"info","ts":"2023-08-
04T00:58:39.530475Z","caller":"snapshot/v3_snapshot.go:97","msg":"saved","path":"/tmp/backup.
db"}
Snapshot saved at /tmp/backup.db
```

Restoring - You can also restore using the etcdctl command line utility. Here, use a different directory to restore from the backup and then change the etcd config file to use that directory.

```
sudo ETCDCTL_API=3 etcdctl --data-dir=/var/lib/restore snapshot restore
/tmp/backup.db
```

NOTE: Now while I was restoring I found that above command is deprecated and the new command that can be used is `etcdutl snapshot restore`

```
sudo ETCDCTL_API=3 etcdutl --data-dir=/var/lib/restore snapshot restore /tmp/backup.db
2023-08-04T01:42:41Z    info    snapshot/v3_snapshot.go:248 restoring snapshot  {"path":
"/tmp/backup.db", "wal-dir": "/var/lib/restore/member/wal", "data-dir": "/var/lib/restore",
"snap-dir": "/var/lib/restore/member/snap", "stack": "go.etcd.io/etcd/etcdutl/v3/snapshot.
(*v3Manager).Restore\n\tgo.etcd.io/etcd/etcdutl/v3/snapshot/v3_snapshot.go:254\ngo.etcd.io/et
cd/etcdutl/v3/etcdutl.SnapshotRestoreCommandFunc\n\tgo.etcd.io/etcd/etcdutl/v3/etcdutl/snapsh
ot_command.go:147\ngo.etcd.io/etcd/etcdutl/v3/etcdutl.snapshotRestoreCommandFunc\n\tgo.etcd.i
o/etcd/etcdutl/v3/etcdutl/snapshot_command.go:117\ngithub.com/spf13/cobra.
(*Command).execute\n\tgithub.com/spf13/cobra@v1.1.3/command.go:856\ngithub.com/spf13/cobra.
(*Command).ExecuteC\n\tgithub.com/spf13/cobra@v1.1.3/command.go:960\ngithub.com/spf13/cobra.
(*Command).Execute\n\tgithub.com/spf13/cobra@v1.1.3/command.go:897\nmain.Start\n\tgo.etcd.io/
etcd/etcdutl/v3/ctl.go:50\nmain.main\n\tgo.etcd.io/etcd/etcdutl/v3/main.go:23\nruntime.main\n
\truntime/proc.go:250"}
2023-08-04T01:42:41Z    info    membership/store.go:141 Trimming membership information from
the backend ...
2023-08-04T01:42:41Z    info    membership/cluster.go:421   added member    {"cluster-id":
"cdf818194e3a8c32", "local-member-id": "0", "added-peer-id": "8e9e05c52164694d", "added-peer-
peer-urls": ["http://localhost:2380"]}
2023-08-04T01:42:41Z    info    snapshot/v3_snapshot.go:269 restored snapshot   {"path":
"/tmp/backup.db", "wal-dir": "/var/lib/restore/member/wal", "data-dir": "/var/lib/restore",
"snap-dir": "/var/lib/restore/member/snap"}
```

```
ls /tmp/restore/
member
```

Now edit the config `vi /etc/kubernetes/manifests/etcd.yaml`

```
    volumeMounts:
    - mountPath: /var/lib/restore
      name: etcd-data
    - mountPath: /etc/kubernetes/pki/etcd
      name: etcd-certs
  hostNetwork: true
  priority: 2000001000
  priorityClassName: system-node-critical
  securityContext:
    seccompProfile:
      type: RuntimeDefault
  volumes:
  - hostPath:
      path: /etc/kubernetes/pki/etcd
      type: DirectoryOrCreate
    name: etcd-certs
  - hostPath:
      path: /var/lib/restore
      type: DirectoryOrCreate
    name: etcd-data
```

Edit the path to `/var/lib/restore` and also the `--data-dir=/var/lib/restore`

As you edit this file, the etcd container will automatically restart, since this is a Static Pod.

Restart kubelet

```
systemctl daemon-reload
systemctl restart kubelet
```

```
kubectl get pods -A | grep etcd
kube-system    etcd-controlplane-a5cc-270721                    1/1
Running   0              2m35s
```

When you describe the pod, you will see that the etcd-data location would have been updated.

Make sure to change the `--data-dir` and the volume changes in the etcd yaml.

This is one of the important scenarios as you need to do all the steps carefully to take the backup and restore it.

Cleanup
Revert all the changes back to `/var/lib/etcd` and wait until etcd becomes healthy again.

Scenario 12 - Create Deployment

Create a deployment with name demo, image=nginx, and replicas 5.

Solution 12-

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

You can simply use the imperative way to create the deployment and specify the number of replicas. Here, the deployment controller will make sure that the desired number of replicas running all times are running by updating the replicaset.

```
kubectl create deploy demo --image=nginx --replicas=5
deployment.apps/demo created
```

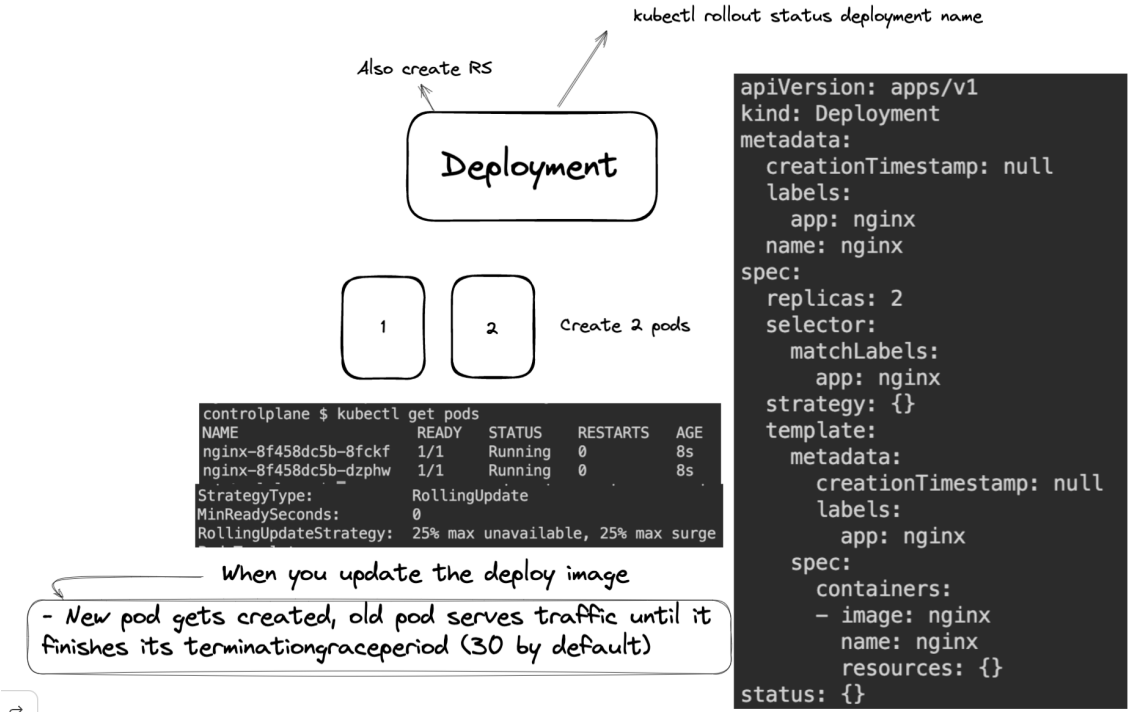
```
controlplane $ kubectl get pods
NAME                    READY   STATUS              RESTARTS   AGE
demo-598ff457d6-5b9pc   0/1     ContainerCreating   0          2s
demo-598ff457d6-6d5pc   0/1     ContainerCreating   0          2s
demo-598ff457d6-nlkzd   1/1     Running             0          2s
demo-598ff457d6-s42vz   0/1     ContainerCreating   0          2s
demo-598ff457d6-t998x   0/1     ContainerCreating   0          2s
controlplane $ kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
demo-598ff457d6-5b9pc   1/1     Running   0          21s
demo-598ff457d6-6d5pc   1/1     Running   0          21s
demo-598ff457d6-nlkzd   1/1     Running   0          21s
demo-598ff457d6-s42vz   1/1     Running   0          21s
demo-598ff457d6-t998x   1/1     Running   0          21s
```

You can see the deployment directly as well and edit that if required to make a change.

```
kubectl get deploy
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
demo   5/5     5            5           3m14s
```

Another thing to note here is, when you create a deployment it will automatically create replicaset for you.

```
kubectl get replicaset
NAME              DESIRED   CURRENT   READY   AGE
demo-598ff457d6   5         5         5       4m30s
```



You can see from the above image that the yaml is to create a deployment with two pods and we will learn about rollout in the next scenario.

When you update any deployment image, the new pod with the new image will get created and the old pod will serve traffic to requests till its **terminatinggraceperiod**.

Also you can change the **Rollingupdatestrategy** to define the minimum % of pods that will be unavailable during an upgrade.

Cleanup

```
kubectl delete deploy demo
deployment.apps "demo" deleted
```

Scenario 13 - Update the image of a deployment.

To create this scenario -> create a deployment with replicas=5 (name=demo and image=nginx) then update the image with nginx:1.22.1.

Solution 13

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster (Kind, Rancher Desktop, etc.)

Create the deployment

```
kubectl create deploy demo --image=nginx --replicas=5
deployment.apps/demo created
```

Image check -> 5 container running with image nginx

```
kubectl get pods   -o jsonpath="{.items[*].spec.containers[*].image}"
|tr -s '[[:space:]]' '\n' |sort |uniq -c
      5 nginx
```

Edit the deployment

```
kubectl edit deploy demo
```

```
spec:
      containers:
      - image: nginx:1.22.1
        imagePullPolicy: Always
        name: nginx
        resources: {}
```

```
root@controlplane-a5cc-270721:~# kubectl get pods
NAME                    READY   STATUS              RESTARTS   AGE
demo-598ff457d6-64s29   1/1     Running             0          3m31s
demo-598ff457d6-rdn4z   1/1     Running             0          3m31s
demo-598ff457d6-wbtpm   1/1     Running             0          3m31s
demo-598ff457d6-zrg79   1/1     Running             0          3m31s
demo-7bd7bf9f9b-7bhpg   0/1     ContainerCreating   0          3s
demo-7bd7bf9f9b-fqds4   0/1     ContainerCreating   0          3s
demo-7bd7bf9f9b-tvx98   0/1     ContainerCreating   0          3s
```

Image check

```
kubectl get pods   -o jsonpath="{.items[*].spec.containers[*].image}"
|tr -s '[[:space:]]' '\n' |sort |uniq -c
      5 nginx:1.21.1
```

As you can see (from the image above), the old pods are terminating and new ones are starting up, and after a few seconds when you do the image check you will see exactly five pods with the newer image.

You can also check this using

```
kubectl rollout status deploy demo
deployment "demo" successfully rolled out
```

Another way to do this is via **imperative** command.

```
kubectl set image deploy demo nginx=nginx:1.21.1
deployment.apps/demo image updated
```

In this way you can update the image for a given deployment.

Cleanup

```
kubectl delete deploy demo
deployment.apps "demo" deleted
```

Scenario 14 - Record and rollback deployment

Create a deployment (name=demo, image=nginx and replicas 4), update the image to nginx:1.22.1 and then revert back to the first revision.

Solution 14 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster(Kind, Rancher Desktop, etc.)

Let's create a deployment first using the imperative command.

```
kubectl create deploy demo --image=nginx --replicas=4
deployment.apps/demo created
```

Now there is a rollout history command that can be used to record any changes made to this deployment. This particular flag helps to know the history and makes it easy to rollback.

```
kubectl rollout history deploy demo
deployment.apps/demo
REVISION  CHANGE-CAUSE
1         <none>
```

Now an interesting thing to note here is that when you create the deployment using imperative command, you cannot add `--record` flag. But if you want to record the first step you can get the output of this imperative command using `--dry-run=client -oyaml` and then create it from the file.

Delete the previous deployment first and run below command:

```
kubectl create deploy demo --image=nginx --replicas=4 --dry-run=client
-oyaml > deploy.yaml
```

```
kubectl create -f deploy.yaml --record
Flag --record has been deprecated, --record will be removed in the
future
deployment.apps/demo created
```

But you need to be aware that the `--record` is deprecated due to this KEP -> https://github.com/kubernetes/enhancements/tree/master/keps/sig-cli/859-kubectl-headers

Even if it gets deprecated, you can annotate it which I will show in the end.
Now, check the rollout history again.

```
kubectl rollout history deploy demo
deployment.apps/demo
```

```
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deploy.yaml --record=true
```

So, you can see now the first revision is not empty.
Now let's update the image as per question to nginx:1.22.1

```
kubectl set image deploy demo nginx=nginx:1.21.1 --record
Flag --record has been deprecated, --record will be removed in the
future
deployment.apps/demo image updated
```

```
kubectl get pods   -o jsonpath="{.items[*].spec.containers[*].image}"
|tr -s '[[:space:]]' '\n' |sort |uniq -c
      4 nginx:1.21.1
```

```
kubectl rollout history deploy demo
deployment.apps/demo
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deploy.yaml --record=true
2         kubectl set image deploy demo nginx=nginx:1.21.1 --record=true
```

Now, Let's say you changed the image and the image had some vulnerabilities or created an issue with your application, you might need to rollback to the previous version.

Let's rollback to revision 1 as stated in the scenario.

```
kubectl rollout undo deploy demo --to-revision=1
deployment.apps/demo rolled back
```

```
kubectl get pods   -o jsonpath="{.items[*].spec.containers[*].image}"
|tr -s '[[:space:]]' '\n' |sort |uniq -c
      4 nginx
```

You can see the deployment got rolled back to revision 1 where the image used was nginx.

Just in case you do not wish to use the `--record` flag , you can annotate.

```
kubectl annotate deploy/demo kubernetes.io/change-cause='update image to
1.21'
deployment.apps/demo annotated
```

```
kubectl rollout history deployment.apps/demo
REVISION  CHANGE-CAUSE
2         kubectl set image deploy demo nginx=nginx:1.21.1 --record=true
3         update image to 1.21
```

You can see when you add the annotation, there is a new revision 3 added with the message that was added with `kubernetes.io/change-cause.`

So this is how you can update deployment, record the changes so that you can easily rollback if something goes wrong.
These are some of the scenarios related to deployments that are useful when you work with Kubernetes. Mostly you will be creating the file and adding more things to it but from CKA's perspective you can make use of the kubectl create command and speed up things using the imperative way.

Cleanup

```
kubectl delete deploy demo
deployment.apps "demo" deleted
```

```
root@controlplane-a5cc-270721:~# kubectl create deploy demo --image=nginx --replicas=4
deployment.apps/demo created
root@controlplane-a5cc-270721:~# kubectl rollout history deploy demo
deployment.apps/demo
REVISION  CHANGE-CAUSE
1         <none>

root@controlplane-a5cc-270721:~# kubectl delete deploy demo
deployment.apps "demo" deleted
root@controlplane-a5cc-270721:~# kubectl create deploy demo --image=nginx --replicas=4 --dry-
run=client -oyaml > deploy.yaml
root@controlplane-a5cc-270721:~# kubectl create -f deploy.yaml --record
Flag --record has been deprecated, --record will be removed in the future
deployment.apps/demo created
root@controlplane-a5cc-270721:~# kubectl rollout history deploy demo
deployment.apps/demo
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deploy.yaml --record=true

root@controlplane-a5cc-270721:~# kubectl set image deploy demo nginx=nginx:1.21.1 --record
Flag --record has been deprecated, --record will be removed in the future
deployment.apps/demo image updated

root@controlplane-a5cc-270721:~# kubectl get pods   -o jsonpath="
{.items[*].spec.containers[*].image}" |tr -s '[[:space:]]' '\n' |sort |uniq -c
      4 nginx:1.21.1
root@controlplane-a5cc-270721:~# kubectl rollout history deploy demo
deployment.apps/demo
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deploy.yaml --record=true
2         kubectl set image deploy demo nginx=nginx:1.21.1 --record=true

root@controlplane-a5cc-270721:~# kubectl rollout undo deploy demo --to-revision=1
deployment.apps/demo rolled back
root@controlplane-a5cc-270721:~# kubectl get pods   -o jsonpath="
{.items[*].spec.containers[*].image}" |tr -s '[[:space:]]' '\n' |sort |uniq -c
      4 nginx
root@controlplane-a5cc-270721:~# kubectl annotate deploy/demo kubernetes.io/change-
cause='update image to 1.21'
deployment.apps/demo annotate
root@controlplane-a5cc-270721:~# kubectl rollout history deploy demodeployment.apps/demo
error: there is no need to specify a resource type as a separate argument when passing
arguments in resource/name form (e.g. 'kubectl get resource/<resource_name>' instead of
'kubectl get resource resource/<resource_name>'
root@controlplane-a5cc-270721:~# kubectl rollout history deployment.apps/demo
deployment.apps/demo
REVISION  CHANGE-CAUSE
2         kubectl set image deploy demo nginx=nginx:1.21.1 --record=true
3         update image to 1.21
```

Scenario 15 - Expose Deployment to the external world

Multiple sub scenarios for this

- Create a deployment name demo and image=nginx
- Expose the deployment to a service type ClusterIP on port 80
- Expose the deployment to a service type NodePort on port 80
- Expose the deployment to a service type ClusterIP on port 3231 with name demo3

Solution 15 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster(Kind, Rancher Desktop, etc.)

Creating a Deployment should be easy now

```
kubectl create deploy demo --image=nginx
deployment.apps/demo created
```

Now when the pods are created, you can access the pods with their IP's and the port but you cannot do that from outside clusters. Also, when the pod dies it gets a new IP address so you cannot use a static IP in your app if you want to access it due to this pod behaviour. Now, to overcome this problem services come to rescue. Services in Kubernetes are a way to expose your application so that you can directly use the service IP and port to access your applications that are running as pods. There are different types of services in Kubernetes:

- ClusterIP - this service IP will be only accessible within the cluster.
- NodePort - You can access this from Node-IP and port from the outside world.
- Loadbalancer - Either cloud specific or projects like global load balancer and this give an external IP assigned that can be used directly anywhere.
- Headless - This is ClusterIP:None and does not get any IP , used for stateful sets.
- Extername - That does not use Labels and selectors but uses direct DNS names.

Now, let's create a clusterIP service in an imperative way.

```
kubectl expose deploy demo --port 80
service/demo exposed
```

```
kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
demo         ClusterIP   10.96.5.52   <none>        80/TCP    14s
```

```
kubectl get svc demo -oyaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2023-08-04T10:34:07Z"
  labels:
    app: demo
  name: demo
  namespace: default
  resourceVersion: "113804"
  uid: 0b97ca19-b987-4aee-b89f-8201d9007290
spec:
  clusterIP: 10.96.5.52
  clusterIPs:
  - 10.96.5.52
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: demo
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

As you can see the cluster type is ClusterIP and it has an IP.
We can use this IP and the port to access the application and it will automatically send traffic to the pod based on the selector `app: demo`
You can also check which pods it will be sending traffic to via the endpoints.

```
root@controlplane-a5cc-270721:~# kubectl get ep demo
NAME   ENDPOINTS        AGE
demo   10.244.2.32:80   106s
root@controlplane-a5cc-270721:~# kubectl get pods -owide
NAME                    READY   STATUS    RESTARTS   AGE     IP            NODE
    NOMINATED NODE   READINESS GATES
demo-598ff457d6-r9xrq   1/1     Running   0          2m57s   10.244.2.32   worker-1-937f-
617615   <none>           <none>
```

You can see that the pod ip and the endpoint IP are the same. This is how the service knows where to send the traffic to.

Now, lets curl the service IP and port:

```
root@controlplane-a5cc-270721:~# curl 10.244.2.32
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

You can see -> Welcome to nginx response! But this can only be accessed inside of a cluster. In order to access the pods from outside of the cluster you can create a service type or **NodePort** or **LoadBalancer**(which is cloud provider specific).

So, let's create a service of type **NodePort**.

```
kubectl expose deploy demo --name=demo2 --port 80 --type=NodePort
service/demo2 exposed
```

Above creates a service named demo2 for deployment demo of type NodePort.

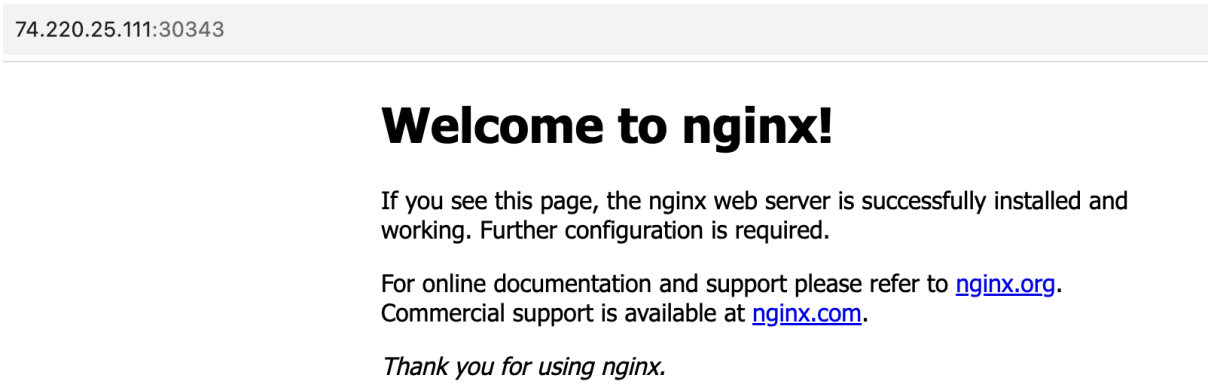
```
kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
demo         ClusterIP   10.96.5.52       <none>        80/TCP         5m19s
demo2        NodePort    10.102.103.169   <none>        80:30343/TCP   22s
```

As you can see, there is **30180** port that can be used with NodeIP to access your app from the outside world. Before that let's just get the yaml file for this as well to see the changes.

When you do `kubectl get svc demo2 -oyaml`
You will see the difference from the previous service as below:

```
  ports:
  - nodePort: 30343
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: demo
  sessionAffinity: None
  type: NodePort
```

Now lets access this from the outside world. I have taken any node Public IP: 30343



Congratulations, you have successfully exported your applications as NodePort to access your application form browser.

For the last part of the question you need to expose it as a service on port 3231

```
kubectl expose deploy demo --name=demo3 --port 3231 --target-port 80
--type ClusterIP
service/demo3 exposed
```

Here we have exposed the service on port 3231 and the target port is the container port that needs to be hit when we call the service on port 3231, that is why we specified 80 as the target port because nginx runs on port 80.

```
kubectl get svc | grep demo3
demo3        ClusterIP   10.99.244.159    <none>        3231/TCP
85s
```

Now let's try to hit the service and port 3231.

```
curl 10.99.244.159:3231
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

You can see that I was able to access it on port 3231.
So in this question you learned Kubernetes services and exposed the deployment as ClusterIP and NodePort with different ports and understood different types of services in Kubernetes as well.

Cleanup

```
kubectl delete deploy demo1
deployment.apps "demo" deleted
```

```
kubectl delete svc demo demo2 demo3
service "demo" deleted
service "demo2" deleted
service "demo3" deleted
```

Services in a bit more diagrammatic way

Scenario 16 - Daemonset

Create a pod that runs on each node with name=ds-demo and image nginx

Solution 16 -

In order to run a pod on every node, there is a Kubernetes object called Daemonset. The most common use case is node-exporter which runs on every node and gathers all the metrics and then used by the Prometheus server. Later, you can create graphs and can easily know the metrics from each node. Now there can be various other use cases like telegraf running agents on each node to collect metrics, connectivity pods, CSI pods etc.

While creating the daemonset you can also set the nodeselector and affinity if you want to place the pods only on certain nodes.
Eg `` `.spec.template.spec.nodeSelector` ``

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster(Kind, Rancher Desktop, etc.)

```
kubectl get nodes
NAME                          STATUS   ROLES           AGE   VERSION
controlplane-a5cc-270721      Ready    control-plane   22h   v1.27.2
worker-1-937f-617615          Ready    <none>          20h   v1.27.2
worker-2-e26c-617615          Ready    <none>          20h   v1.27.2
```

Create a file called `ds.yaml` as there is no way currently to create a daemonset via the imperative way.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: ds-demo
  labels:
    k8s-app: ds-demo
spec:
  selector:
    matchLabels:
      name: ds-demo
  template:
    metadata:
      labels:
        name: ds-demo
    spec:
```

```
      tolerations:
      # these tolerations are to have the daemonset runnable on control
plane nod
      # remove them if your control plane nodes should not run pods
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      containers:
      - name: ds-dmeo
        image: nginx
```

In the above yaml, you can see that the tolerations have been defined which means this Daemonset when created will be able to create the pods on all the three nodes including the controlplane node.

Apply the DaemonSet manifest.

```
root@controlplane-a5cc-270721:~# kubectl apply -f ds.yaml
daemonset.apps/ds-demo created
root@controlplane-a5cc-270721:~# kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
ds-demo-6lr7d   1/1     Running   0          3s
ds-demo-hshkq   1/1     Running   0          3s
ds-demo-vj6lr   1/1     Running   0          3s
root@controlplane-a5cc-270721:~# kubectl get pods -owide
NAME            READY   STATUS    RESTARTS   AGE   IP            NODE
NOMINATED NODE   READINESS GATES
ds-demo-6lr7d   1/1     Running   0          8s    10.244.0.16   controlplane-a5cc-270721
<none>           <none>
ds-demo-hshkq   1/1     Running   0          8s    10.244.1.27   worker-2-e26c-617615
<none>           <none>
ds-demo-vj6lr   1/1     Running   0          8s    10.244.2.33   worker-1-937f-617615
<none>           <none>
```

You can see, all the nodes have a copy of nginx pod running on them.

Cleanup

```
kubectl delete -f ds.yaml
daemonset.apps "ds-demo" deleted
```

Scenario 17 - Persistent Volume and Persistent Volume claim

Create a persistent volume claim `saiyam` with the following details.
Access mode: RWO
Capacity : 2Gi
Storageclass: longhorn
Use that pvc in a new pod `ksctl` with nginx image, with the volume mounted at `/usr/share/nginx/html/index.html`

Solution 17 -

Often comes the need to persist data and when you create a normal pod, you cannot persist state. In Kubernetes there is a concept of persistent volume and persistent volume claim. PV is the storage created by the admin or dynamically by the storage class. PV will carry the details of actual storage and as an end user/developer, they will be creating a Kubernetes object called PVC. Now as soon as the PVC is created, that particular PV is claimed. Depending on the use case, users will request for a storage with different properties and based on what cluster admins have configured, they will be able to claim a volume.
Using this concept, even if the pod goes down and gets recreated, the data remains, as this data has a separate life cycle than the pod lifecycle using this. There can be external storage also attached which do not live on the nodes like NFS etc or cloud specific, making it more robust if even the cluster goes down, the storage will still remain and the date will still be persisted.

Coming to the solution, lets first create the scenario, for this scenario we will install the most basic local path provisioner by rancher so that we get the storage class and then create the PVC and pod accordingly.

Local path provisioner install

```
kubectl apply -f
https://raw.githubusercontent.com/rancher/local-path-provisioner/v0.0.24
/deploy/local-path-storage.yaml
```

Make it default storage class

```
kubectl patch storageclass local-path -p '{"metadata":
{"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
storageclass.storage.k8s.io/local-path patched
```

```
root@controlplane-a5cc-270721:~# kubectl get sc
No resources found
root@controlplane-a5cc-270721:~# kubectl apply -f
https://raw.githubusercontent.com/rancher/local-path-provisioner/v0.0.24/deploy/local-path-
storage.yaml
namespace/local-path-storage created
serviceaccount/local-path-provisioner-service-account created
clusterrole.rbac.authorization.k8s.io/local-path-provisioner-role created
clusterrolebinding.rbac.authorization.k8s.io/local-path-provisioner-bind created
deployment.apps/local-path-provisioner created
storageclass.storage.k8s.io/local-path created
configmap/local-path-config created
root@controlplane-a5cc-270721:~# kubectl get sc
NAME         PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
local-path   rancher.io/local-path   Delete          WaitForFirstConsumer   false
     22s
```

Create persistent volume claim
Create a file `pv.yaml` with below contents:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: saiyam
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

Also create a pod.yaml file with below contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: ksctl
spec:
  volumes:
    - name: pv-saiyam
      persistentVolumeClaim:
        claimName: saiyam
  containers:
    - name: ksctl
      image: nginx
      ports:
        - containerPort: 80
          name: "http"
```

```
    volumeMounts:
      - mountPath: "/usr/share/nginx/html"
        name: pv-saiyam
```

Let's try to understand first what is happening, in the pvc you are specifying how much storage you need, what type of mode you need and the name. Once you apply this, automatically the storage class will create the PV but in this case when you do `kubectl get sc` Volumebindingmode is WaitForFirstConsumer. This means it will create the volume as soon as you use the PVC in the pod.
From the pod.yaml file you can see that there are two important sections:

- Volumes - this is where we define persistentVolumeClaim with claimName
- volumeMounts - where inside the pod this claim should be mounted to.

```
root@controlplane-a5cc-270721:~ #kubectl apply -f pvc.yaml
persistentvolumeclaim/saiyam created
root@controlplane-a5cc-270721:~# kubectl get pvc
NAME     STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
saiyam   Pending                                                     10s
root@controlplane-a5cc-270721:~# kubectl get sc
NAME         PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
local-path   rancher.io/local-path   Delete          WaitForFirstConsumer   false
     5h34m
```

Let's create the pod

```
root@controlplane-a5cc-270721:~# kubectl apply -f pod.yaml
pod/ksctl created
root@controlplane-a5cc-270721:~# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
  CLAIM            STORAGECLASS   REASON   AGE
pvc-03b4cf2d-82f9-4d14-aa2a-9948aa01b821   2Gi        RWO            Delete           Bound
default/saiyam   local-path              62s
root@controlplane-a5cc-270721:~# kubectl get pvc
NAME     STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
saiyam   Bound    pvc-03b4cf2d-82f9-4d14-aa2a-9948aa01b821   2Gi        RWO            local-
path     12m
root@controlplane-a5cc-270721:~# kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
ksctl   1/1     Running   0          5m11s
```

As soon as you create the pod, pv will be created and then bound to the pvc and the pod will go in the running state.

WAIT A MINUTE! We did not specify the longhorn storage class!
Since we did not specify any storage class so it took the default storage class, remember in one of the steps above there was a step to make local-path as the default storage class. In order to use the default storage class, let's install longhorn first.

Install longhorn

```
kubectl apply -f
https://raw.githubusercontent.com/longhorn/longhorn/v1.5.1/deploy/longho
rn.yaml
```

Longhorn is a production ready storage engine so it will create a bunch of things when applied onto the Kubernetes cluster.

```
root@controlplane-a5cc-270721:~# kubectl apply -f
https://raw.githubusercontent.com/longhorn/longhorn/v1.5.1/deploy/longhorn.yaml
namespace/longhorn-system created
serviceaccount/longhorn-service-account created
serviceaccount/longhorn-support-bundle created
configmap/longhorn-default-setting created
configmap/longhorn-storageclass created
customresourcedefinition.apiextensions.k8s.io/backingimagedatasources.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/backingimagemanagers.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/backingimages.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/backups.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/backuptargets.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/backupvolumes.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/engineimages.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/engines.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/instancemanagers.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/nodes.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/orphans.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/recurringjobs.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/replicas.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/settings.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/sharemanagers.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/snapshots.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/supportbundles.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/systembackups.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/systemrestores.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/volumes.longhorn.io created
customresourcedefinition.apiextensions.k8s.io/volumeattachments.longhorn.io created
clusterrole.rbac.authorization.k8s.io/longhorn-role created
clusterrolebinding.rbac.authorization.k8s.io/longhorn-bind created
clusterrolebinding.rbac.authorization.k8s.io/longhorn-support-bundle created
service/longhorn-backend created
service/longhorn-frontend created
service/longhorn-conversion-webhook created
service/longhorn-admission-webhook created
service/longhorn-recovery-backend created
service/longhorn-engine-manager created
service/longhorn-replica-manager created
daemonset.apps/longhorn-manager created
deployment.apps/longhorn-driver-deployer created
deployment.apps/longhorn-ui created
```

Check the storage classes

```
kubectl get sc
NAME                 PROVISIONER             RECLAIMPOLICY
VOLUMEBINDINGMODE    ALLOWVOLUMEEXPANSION   AGE
local-path (default)   rancher.io/local-path   Delete
```

```
WaitForFirstConsumer   false                  5h52m
longhorn (default)     driver.longhorn.io      Delete          Immediate
true                   66s
```

You can see both the classes have default written, So if we do not specify anything and repeat the scenario, it will use the latest default storage class which is Longhorn.

Let's delete the pvc and pod

```
root@controlplane-a5cc-270721:~# kubectl delete pod ksctl
pod "ksctl" deleted
root@controlplane-a5cc-270721:~# kubectl delete pvc saiyam
persistentvolumeclaim "saiyam" deleted
root@controlplane-a5cc-270721:~# kubectl get pv
No resources found
```

In the `pvc.yaml` file add `storageClassName: longhorn`

```
cat pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: saiyam
spec:
  storageClassName: longhorn
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

Create the pvc and pod

```
root@controlplane-a5cc-270721:~# kubectl apply -f pvc.yaml
persistentvolumeclaim/saiyam created
root@controlplane-a5cc-270721:~# kubectl get pvc
NAME     STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
saiyam   Bound    pvc-95ce5bd6-8b9f-47c2-8268-bc53aa02504d   2Gi        RWO
longhorn       4s
root@controlplane-a5cc-270721:~# kubectl get sc
NAME                 PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
local-path (default)   rancher.io/local-path   Delete          WaitForFirstConsumer   false
               5h59m
longhorn (default)     driver.longhorn.io      Delete          Immediate              true
                8m28s
root@controlplane-a5cc-270721:~# kubectl apply -f pod.yaml
pod/ksctl created
root@controlplane-a5cc-270721:~# kubectl get pod
NAME    READY   STATUS              RESTARTS   AGE
ksctl   0/1     ContainerCreating   0          5s
root@controlplane-a5cc-270721:~# kubectl get pod
NAME    READY   STATUS    RESTARTS   AGE
ksctl   1/1     Running   0          15s
```

You can see that the volumebinding mode is immediate, so as soon as the pvc got created, the pv got created and was automatically bound to that pvc without needing to create the pod. So this should clear the volumebindingmode concept as well.

You can also see the storageclass mentioned in the pvc in longhorn. So this completes the scenario.

There was a lot covered in this scenario, multiple storage classes, volumebindingmodes, making storageclass as default and how to create a pod and use the concept of pvc for its volume.

Cleanup

```
root@controlplane-a5cc-270721:~# kubectl delete pod ksctl
pod "ksctl" deleted
root@controlplane-a5cc-270721:~# kubectl delete pvc saiyam
persistentvolumeclaim "saiyam" deleted
```

Scenario 18 - Multi-container shared volume

Create a pod with three containers
c1 executes the command `"echo "Hello from c1 container."` and save to file `/var/log/newfile`
c2 executes the command `"echo "Hello from c1 container."` and save to file `/var/log/newfile`
c3 executes the command `"cat var/log/newfile"`
Make sure you make use of volumes and volumemounts for making path `/var/log/` available to all containers. Also make sure that the pod does not CrashLoopBackOff.

Solution 18:

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster(Kind, Rancher Desktop, etc.)

The tricky part here is the shared volume and for that you can use an `emptyDir` volume.EmptyDir volume gets created as soon as the pod is assigned to the node and it stays throughout the lifespan of the pod. It gets removed once the pod is deleted. You can use DISK or RAM as the volume. By default it's the DISK.

For this particular scenario, create a `pod.yaml` file with below contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  volumes:
  - name: log-volume
    emptyDir: {}
  restartPolicy: OnFailure
  containers:
  - name: c1
    image: busybox
    command:
    - sh
    - -c
    - 'echo "Hello from c1 container." | tee -a /var/log/newfile'
    volumeMounts:
    - name: log-volume
      mountPath: /var/log

  - name: c2
    image: busybox
    command:
    - sh
    - -c
```

```
    - 'echo "Hello from c2 container." | tee -a /var/log/newfile'
    volumeMounts:
    - name: log-volume
      mountPath: /var/log

  - name: c3
    image: busybox
    command:
    - sh
    - -c
    - 'cat /var/log/newfile'
    volumeMounts:
    - name: log-volume
      mountPath: /var/log
```

In the above pod manifest file you can see the volume mount is used for all the containers and there is a single volume which is the emptyDir. The `restartPolicy: OnFailure` Do not restart the pod after completion and the pod will not go into crashloopbackoff. Apply the manifest and then you can see the logs from all the containers.

```
# kubectl apply -f pod.yaml
pod/my-pod created
# kubectl get pods
NAME     READY   STATUS              RESTARTS   AGE
my-pod   0/3     ContainerCreating   0          2s
# kubectl get pods
NAME     READY   STATUS      RESTARTS   AGE
my-pod   0/3     Completed   0          4s
# kubectl logs my-pod -c c1
Hello from c1 container.
# kubectl logs my-pod -c c2
Hello from c2 container.
# kubectl logs my-pod -c c3
Hello from c1 container.
Hello from c2 container.
```

Cleanup

```
kubectl delete -f pod.yaml
pod "my-pod" deleted
```

Scenario 19 - Troubleshooting: Logs

Case 1 - Find the logs of the pod named demo and save that to `/tmp/mylog.txt` file.
Case 2 - Create a multiple container pod with multiple containers names c1,c2 using nginx and redis image respectively and save the logs of container c2 to `/tmp/c2.txt` file.

Solution 19 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster(Kind, Rancher Desktop, etc.)

Create a pod names demo with any image, let's say nginx for this case.

```
kubectl run demo --image=nginx
```

```
root@controlplane-a5cc-270721:~# kubectl run demo --image=nginx
pod/demo created
root@controlplane-a5cc-270721:~# kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
demo   1/1     Running   0          3s
```

Now in Kubernetes, you can view logs from the containers by using the logs command.
Some common logs command

- `kubectl logs demo`: to get the logs of the pod.
- `kubectl logs -f demo`: to follow the logs for the pod.
- `kubectl logs demo --all-containers=true:` to get logs from all containers.
- `kubectl logs demo -c <containername>`: to get logs from a specific container in case of multi-container pod.
- `kubectl logs -l app=nginx --all-containers=true`: logs form all containers in a pod with a specified label which in this case is `app:nginx`
- `kubectl logs -c c1 -p demo`: This will give logs from previously terminated container c1 from the demo pod. Useful when you are troubleshooting the crassloopbackoff error.
- `kubectl logs --tail=10 demo`: get most recent 10 line log output from the demo pod.
- `kubectl logs demo --all-containers=true --prefix=true`: to get logs with the log source.

Now for the pod above created you can simply view the logs by `kubectl logs demo` and then put that into the specified file as per the scenario using the `> operator`.

```
root@controlplane-a5cc-270721:~# kubectl logs demo
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in
/etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/08/07 03:43:41 [notice] 1#1: using the "epoll" event method
2023/08/07 03:43:41 [notice] 1#1: nginx/1.25.1
2023/08/07 03:43:41 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2023/08/07 03:43:41 [notice] 1#1: OS: Linux 5.15.0-40-generic
2023/08/07 03:43:41 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/08/07 03:43:41 [notice] 1#1: start worker processes
2023/08/07 03:43:41 [notice] 1#1: start worker process 29
2023/08/07 03:43:41 [notice] 1#1: start worker process 30

root@controlplane-a5cc-270721:~# kubectl logs demo > /tmp/mylog.txt
root@controlplane-a5cc-270721:~# cat /tmp/mylog.txt
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in
/etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/08/07 03:43:41 [notice] 1#1: using the "epoll" event method
2023/08/07 03:43:41 [notice] 1#1: nginx/1.25.1
2023/08/07 03:43:41 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2023/08/07 03:43:41 [notice] 1#1: OS: Linux 5.15.0-40-generic
2023/08/07 03:43:41 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/08/07 03:43:41 [notice] 1#1: start worker processes
2023/08/07 03:43:41 [notice] 1#1: start worker process 29
2023/08/07 03:43:41 [notice] 1#1: start worker process 30
```

For case 2, delete the first pod and create the pod with the below manifest.

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: demo
  name: demo
spec:
  containers:
  - image: nginx
    name: c1
```

```
    resources: {}
  - image: redis
    name: c2
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Now, apply the manifest, get the logs of container using `kubectl logs demo -c c2` and use `>` to store to a file mentioned in the scenario `/tmp/c2.txt`

```
root@controlplane-a5cc-270721:~# kubectl create -f pod.yaml
pod/demo created
root@controlplane-a5cc-270721:~# kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
demo   2/2     Running   0          5s

root@controlplane-a5cc-270721:~# kubectl logs demo -c c2
1:C 07 Aug 2023 05:15:59.556 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
1:C 07 Aug 2023 05:15:59.557 # Redis version=7.0.12, bits=64, commit=00000000, modified=0,
pid=1, just started
1:C 07 Aug 2023 05:15:59.557 # Warning: no config file specified, using the default config.
In order to specify a config file use redis-server /path/to/redis.conf
1:M 07 Aug 2023 05:15:59.559 * monotonic clock: POSIX clock_gettime
1:M 07 Aug 2023 05:15:59.566 * Running mode=standalone, port=6379.
1:M 07 Aug 2023 05:15:59.566 # Server initialized
1:M 07 Aug 2023 05:15:59.569 * Ready to accept connections

root@controlplane-a5cc-270721:~# kubectl logs demo -c c2 > /tmp/c2.txt
root@controlplane-a5cc-270721:~# cat /tmp/c2.txt
1:C 07 Aug 2023 05:15:59.556 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
1:C 07 Aug 2023 05:15:59.557 # Redis version=7.0.12, bits=64, commit=00000000, modified=0,
pid=1, just started
1:C 07 Aug 2023 05:15:59.557 # Warning: no config file specified, using the default config.
In order to specify a config file use redis-server /path/to/redis.conf
1:M 07 Aug 2023 05:15:59.559 * monotonic clock: POSIX clock_gettime
1:M 07 Aug 2023 05:15:59.566 * Running mode=standalone, port=6379.
1:M 07 Aug 2023 05:15:59.566 # Server initialized
1:M 07 Aug 2023 05:15:59.569 * Ready to accept connections
```

These are just a couple of scenarios, try out all the types of logs commands mentioned above. Below is the last example where you have the prefix as true to get the pod/container info with each log message.

```
kubectl logs demo --all-containers=true --prefix=true
[pod/demo/c1] /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to
perform configuration
[pod/demo/c1] /docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
[pod/demo/c1] /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-
default.sh
......modified=0, pid=1, just started
[pod/demo/c2] 1:C 07 Aug 2023 05:15:59.557 # Warning: no config file specified, using the
default config. In order to specify a config file use redis-server /path/to/redis.conf
[pod/demo/c2] 1:M 07 Aug 2023 05:15:59.559 * monotonic clock: POSIX clock_gettime
.....
```

You can see in the highlighted section about the pod/container details from where the logs are coming.

Cleanup

```
kubectl delete -f pod.yaml
pod "demo" deleted
```

Scenario 20 - Troubleshooting - Node not ready

Fix the Node that is in the NotReady state

Solution 20 -

For this particular scenario, there can be various ways due to which the node can be in a not ready state but what I will try to show you here is, I will be creating some failure scenarios and then fix those. So to understand better, you will learn different places to check when a question like this appears and you know different ways or different types of checks that you can do to check why the node is in a NotReady status.

First, I will create the scenario and try to fix it. Finally, I will tell you how to create that scenario. While fixing I will show what all things and which all places to check for.

I have a node in below state

```
kubectl get nodes
NAME                          STATUS     ROLES           AGE     VERSION
controlplane-a5cc-270721      Ready      control-plane   4d      v1.27.2
worker-1-937f-617615          NotReady   <none>          3d21h   v1.27.2
worker-2-e26c-617615          Ready      <none>          3d21h   v1.27.1
```

The first troubleshooting step in Kubernetes, whether its a deployment, pod or a node that is not working is to describe it and see the events and other related signals to give you a hint what is wrong.
In the above case, you can see that worker-1 node is not ready, so let's describe it.

```
kubectl describe node worker-1-937f-617615
```

```
Conditions:
  Type                 Status    LastHeartbeatTime                 LastTransitionTime
       Reason              Message
  ----                 ------    -----------------                 ------------------
       ------              -------
  NetworkUnavailable   False     Thu, 03 Aug 2023 15:40:03 +0000   Thu, 03 Aug 2023 15:40:03
+0000   FlannelIsUp          Flannel is running on this node
  MemoryPressure       Unknown   Mon, 07 Aug 2023 10:34:07 +0000   Mon, 07 Aug 2023 10:38:56
+0000   NodeStatusUnknown    Kubelet stopped posting node status.
  DiskPressure         Unknown   Mon, 07 Aug 2023 10:34:07 +0000   Mon, 07 Aug 2023 10:38:56
+0000   NodeStatusUnknown    Kubelet stopped posting node status.
  PIDPressure          Unknown   Mon, 07 Aug 2023 10:34:07 +0000   Mon, 07 Aug 2023 10:38:56
+0000   NodeStatusUnknown    Kubelet stopped posting node status.
  Ready                Unknown   Mon, 07 Aug 2023 10:34:07 +0000   Mon, 07 Aug 2023 10:38:56
+0000   NodeStatusUnknown    Kubelet stopped posting node status.
```

From the image, you can see that kubelet has stopped posting the node status, which means there is some issue with the kubelet. Since the kubelet of worker-1 is not healthy, the next step would be to login to the worker node and check what is wrong with the kubelet.

Login to the worker-1 node and run `systemctl status kubelet`

```
● kubelet.service - kubelet: The Kubernetes Node Agent
     Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
    Drop-In: /etc/systemd/system/kubelet.service.d
             └─10-kubeadm.conf
     Active: active (running) since Mon 2023-08-07 10:38:18 UTC; 3h 1min ago
       Docs: https://kubernetes.io/docs/home/
   Main PID: 3704810 (kubelet)
      Tasks: 11 (limit: 4355)
     Memory: 51.7M
        CPU: 4min 19.031s
     CGroup: /system.slice/kubelet.service
             └─3704810 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-
kubelet.co>
```

Kubelet status is running, sometimes you will be able to see the errors from this command itself. In this case also by running this command I am able to see a clue on where to go next.

```
r.go:262] "Eviction manager: failed to get summary stats" err="failed to get node info: node
\"worker-1-937f-617615\" not found"
3] vendor/k8s.io/client-go/informers/factory.go:150: failed to list *v1.CSIDriver: Get
"https://controlplane-a5cc-270721:64444443/apis/storage.k8s.io/v1/csidrivers?
limit=500&resourceVersion=0": dial tcp: address 64444443: invalid port
8] vendor/k8s.io/client-go/informers/factory.go:150: Failed to watch *v1.CSIDriver: failed to
list *v1.CSIDriver: Get "https://controlplane-a5cc-
270721:64444443/apis/storage.k8s.io/v1/csidrivers?limit=500&resourceVersion=0": dial tcp:
address 64444443: invalid port
46] "Failed to ensure lease exists, will retry" err="Get \"https://controlplane-a5cc-
270721:64444443/apis/coordination.k8s.io/v1/namespaces/kube-node-lease/leases/worker-1-937f-
617615?timeout=10s\": dial tcp: address 64444443: invalid port" interval="7s"
```

Looks like the port on which it's trying to connect on the controlplane is not correct.
On the controlplane side use command `kubectl config view` to find.

```
kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://controlplane-a5cc-270721:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
```

You can see that the port should have been 6443 and not 6444443. LEt’s fix this.

But wait, where to fix this ? Again see the kubelet service to see from where it is loading it.
It is `cat /etc/systemd/system/kubelet.service.d/10-kubeadm.conf`

```
cat /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
# Note: This dropin only works with kubeadm and kubelet v1.11+
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-
kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf"
Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml"
# This is a file that "kubeadm init" and "kubeadm join" generates at runtime, populating the
KUBELET_KUBEADM_ARGS variable dynamically
EnvironmentFile=-/var/lib/kubelet/kubeadm-flags.env
# This is a file that the user can use for overrides of the kubelet args as a last resort.
Preferably, the user should use
# the .NodeRegistration.KubeletExtraArgs object in the configuration files instead.
KUBELET_EXTRA_ARGS should be sourced from this file.
EnvironmentFile=-/etc/default/kubelet
ExecStart=
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS
$KUBELET_KUBEADM_ARGS $KUBELET_EXTRA_ARGS
```

From here we know that the next file we need to see is `/etc/kubernetes/kubelet.conf`

```
root@worker-1-937f-617615:~# vi /etc/kubernetes/kubelet.conf

    certificate-authority-data: xxxxxxx
    server: https://controlplane-a5cc-270721:64444443
  name: default-cluster
contexts:
- context:
    cluster: default-cluster
    namespace: default
    user: default-auth
  name: default-context
current-context: default-context
kind: Config
preferences: {}
users:
- name: default-auth
  user:
    client-certificate: /var/lib/kubelet/pki/kubelet-client-current.pem
    client-key: /var/lib/kubelet/pki/kubelet-client-current.pem
```

As you can see, the server is having the wrong port, so I changed to 6443 and saved the file.
Note here that, there can be other places like the key, certificate location that might be wrong as well.
Once you save, do `systemctl daemon-reload` and s`ystemctl restart kubelet`
You will be able to see that all nodes become ready.

```
kubectl get nodes
```

```
NAME                      STATUS   ROLES           AGE     VERSION
controlplane-a5cc-270721  Ready    control-plane   4d1h    v1.27.2
worker-1-937f-617615      Ready    <none>          3d22h   v1.27.2
worker-2-e26c-617615      Ready    <none>          3d22h   v1.27.1
```

In order to create this scenario, just go to the worker node, edit `/etc/kubernetes/kubelet.conf` file and change address/port of the server or cert location etc and the kubelet will stop posting ready status, node becomes notReady.

Let's try one more troubleshooting:

```
root@controlplane-a5cc-270721:/# kubectl get nodes
The connection to the server controlplane-a5cc-270721:6443 was refused -
did you specify the right host or port?
```

Right now my controlplane itself is not reachable, this means I am not able to communicate with the api-server. In this case the first initial step would be to look at one folder which is `/etc/kubernetes/manifests`. In this folder, there are multiple files that are responsible for running the control plane components. Like the kube-apiserver pod that runs, controller manager pod and other static pods. This directory is configured in the kubelet config and it picks up the files from this directory and runs them as pods.
Since I am not able to communicate with the api-server itself, I will try to see if anything is wrong with the `kube-apiserver.yaml` file.

```
root@controlplane-a5cc-270721:/etc/kubernetes/manifests# ls
etcd.yaml  kube-apiserver.yaml  kube-controller-manager.yaml
kube-scheduler.yaml
```

```yaml
containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.1.2
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pkii/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
```

When I opened the file, I saw a tricky one here -> pkii instead of pki. I fixed it and the api-server got back up and the nodes looked in a healthy and Ready state again.

```
kubectl get nodes
NAME                      STATUS   ROLES           AGE    VERSION
controlplane-a5cc-270721  Ready    control-plane   4d2h   v1.27.2
worker-1-937f-617615      Ready    <none>          4d     v1.27.2
worker-2-e26c-617615      Ready    <none>          4d     v1.27.1
```

In order to replicate the above scenario, you just need to edit the `/etc/kubernetes/manifests/kube-apiserver.yaml` and make the change of pki to pkii or any other change so that the node can fail. Then you sleep for a day and next morning debug the cluster 😀

Like this, there can be many troubleshooting troubleshooting parts that you might want to look at when you have either problems with your workloads or with your clusters itself. Let's recap some of the things that you can try to find and point where the issue is.

You can check the Kubernetes troubleshooting workshop that I did with my friend David to give you a more deep dive on Kubernetes troubleshooting as this is a really deep topic and there are many ways to troubleshoot:

- Describe the pod/deployment/service or any kubernetes object.
- Kubectl get events: stored for a limited time depending on cluster config, Also there are other variations to get more understandable output `kubectl get events --sort-by='.lastTimestamp'`,

```
kubectl get events --sort-by='.lastTimestamp'
LAST SEEN   TYPE     REASON           OBJECT                              MESSAGE
3m34s       Normal   RegisteredNode   node/controlplane-a5cc-270721   Node controlplane-a5cc-
270721 event: Registered Node controlplane-a5cc-270721 in Controller
3m34s       Normal   RegisteredNode   node/worker-1-937f-617615       Node worker-1-937f-
617615 event: Registered Node worker-1-937f-617615 in Controller
3m34s       Normal   RegisteredNode   node/worker-2-e26c-617615       Node worker-2-e26c-
617615 event: Registered Node worker-2-e26c-617615 in Controller

kubectl get events --field-selector reason=Started -A
NAMESPACE     LAST SEEN   TYPE     REASON    OBJECT
MESSAGE
kube-system   4m16s       Normal   Started   pod/kube-apiserver-controlplane-a5cc-270721
Started container kube-apiserver
```

- View the logs for deployment/pod: I have explained that more in the logs scenario
- For cluster troubleshooting, check kubelet status and try to get info from there, see the `/etc/kubernetes/manifests` file and see if everything is correct, check kubelet config.

There are many scenarios and covering all is out of scope for this book but this should give you a good starting point and also some areas to look out for when things are not working out.

Scenario 21 - Network Policy

- Create a cluster with NetworkPolicy enabled - Cilium
- Create two namespaces dev1 and dev2
- Create a pod demo with nginx image and run in the dev1 namespace
- Create a pod demo-dev2 with nginx image and run in the dev2 namespace
- Connect from demo pod to demo-dev2
- Create a Network policy deny-all for the pods running in dev2 namespace where no traffic can reach to the pods in dev2 namespace
- Create a Network Policy kube-demo that allows ingress traffic from pods in dev1 namespace can connect to the pods in the dev2 namespace over port 80.

Solution 21 -

First, I will create the scenario and try to fix it. Finally, I will tell you how to create that scenario. While fixing I will show what all things and which all places to check for.
But in order to process the network policy scenario you need the CNI layer to be Calico, Cilium, or Weave Net. As of now the current CNI is Flannel, so let's remove Flannel and install Cilium.

```
kubectl delete -f
https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel.
yml
```

Install Cilium cli

```
CILIUM_CLI_VERSION=$(curl -s
https://raw.githubusercontent.com/cilium/cilium-cli/main/stable.txt)
CLI_ARCH=amd64
if [ "$(uname -m)" = "aarch64" ]; then CLI_ARCH=arm64; fi
curl -L --fail --remote-name-all
https://github.com/cilium/cilium-cli/releases/download/${CILIUM_CLI_VERS
ION}/cilium-linux-${CLI_ARCH}.tar.gz{,.sha256sum}
sha256sum --check cilium-linux-${CLI_ARCH}.tar.gz.sha256sum
sudo tar xzvfC cilium-linux-${CLI_ARCH}.tar.gz /usr/local/bin
rm cilium-linux-${CLI_ARCH}.tar.gz{,.sha256sum}
```

Install Cilium

```
cilium install --version 1.14.0
```

Check status of Cilium installation

```
cilium status --wait
```

```
root@controlplane-a5cc-270721:/# cilium install --version 1.14.0
ℹ️  Using Cilium version 1.14.0
🔮 Auto-detected cluster name: kubernetes
🔮 Auto-detected datapath mode: tunnel
🔮 Auto-detected kube-proxy has been installed
root@controlplane-a5cc-270721:/# cilium status --wait
    /¯¯\
 /¯¯\__/¯¯\    Cilium:             OK
 \__/¯¯\__/    Operator:           OK
 /¯¯\__/¯¯\    Envoy DaemonSet:    disabled (using embedded mode)
 \__/¯¯\__/    Hubble Relay:       disabled
    \__/       ClusterMesh:        disabled

Deployment             cilium-operator    Desired: 1, Ready: 1/1, Available: 1/1
DaemonSet              cilium             Desired: 3, Ready: 3/3, Available: 3/3
Containers:            cilium             Running: 3
                       cilium-operator    Running: 1
Cluster Pods:          2/30 managed by Cilium
Helm chart version:    1.14.0
Image versions         cilium
quay.io/cilium/cilium:v1.14.0@sha256:5a94b561f4651fcfd85970a50bc78b201cfbd6e2ab1a03848eab25a8
2832653a: 3
                       cilium-operator    quay.io/cilium/operator-
generic:v1.14.0@sha256:3014d4bcb8352f0ddef90fa3b5eb1bbf179b91024813a90a0066eb4517ba93c9: 1
```

Now, the cluster is ready to take up the scenario.
Network policy is a concept that enables the user's to allow/deny traffic from/to other pods. It can be a strong isolation and lead up to good security practice.

This is a very interesting scenario as at each step I will apply a network policy and then check the access.

Firstly, create both namespaces and both the pods in that namespaces as mentioned in the scenario.

```
kubectl create ns dev1
kubectl create ns dev2
kubectl run demo --image=nginx -n dev1
kubectl run demo-dev2 --image=nginx -n dev2
```

```
kubectl create ns dev1
namespace/dev1 created
kubectl create ns dev2
namespace/dev2 created
kubectl run demo --image=nginx -n dev1
pod/demo created
kubectl run demo-dev2 --image=nginx -n dev2
pod/demo-dev2 created
kubectl get pod -owide -n dev1
NAME   READY   STATUS    RESTARTS   AGE   IP           NODE                   NOMINATED NODE
  READINESS GATES
demo   1/1     Running   0          19s   10.0.2.221   worker-1-937f-617615   <none>
  <none>
kubectl get pod -owide -n dev2
NAME        READY   STATUS    RESTARTS   AGE   IP          NODE                   NOMINATED
NODE   READINESS GATES
demo-dev2   1/1     Running   0          20s   10.0.2.57   worker-1-937f-617615   <none>
      <none>
```

Now connect from demo to demo-dev2

```
kubectl exec -it demo -n dev1 -- curl 10.0.2.57:80
```

```
kubectl exec -it demo -n dev1 -- curl 10.0.2.57:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
```

The connection works fine.
Create the `deny_all.yaml` network policy with below content.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: dev2
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Above is a NetworkPolicy that is applied to all pods in the dev2 namespace because the podSelector is empty braces. The policy applied is the ingress policy and since I have not defined any ingress rules. Hence this policy effectively denies all incoming traffic to all pods in the dev2 namespace.

Let's rerun the connect command again -> `kubectl exec -it demo -n dev1 -- curl 10.0.2.57:80`

```
root@controlplane-a5cc-270721:/# kubectl apply -f deny_all.yaml
networkpolicy.networking.k8s.io/deny-all created
root@controlplane-a5cc-270721:/# kubectl exec -it demo -n dev1 -- curl 10.0.2.57:80

curl: (28) Failed to connect to 10.0.2.57 port 80 after 129992 ms: Couldn't connect to server
command terminated with exit code 28
```

Now, let's create the Network policy that allows ingress traffic from pods in dev1 namespace can connect to the pods in the dev2 namespace over port 80.
Create a file `np.yaml` with below contents:
apiVersion: networking.k8s.io/v1

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: kube-demo
  namespace: dev2
spec:
  podSelector: {}
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: dev1
    ports:
    - protocol: TCP
      port: 80
```

Here in the `policyType` Ingress, the ingress is allowed from the pods in namespace dev1 using the `namespaceSelector.`

Delete the previous `NetworkPolicy` of `deny-all` and apply the new one or you can also edit the same one.
Let's run the same command and it should work now as the traffic from the pod demo in dev1 namespace would be allowed to the pods in dev2 namespace.

```
root@controlplane-a5cc-270721:/# kubectl apply -f np.yaml
networkpolicy.networking.k8s.io/kube-demo created
root@controlplane-a5cc-270721:/# kubectl exec -it demo -n dev1 -- curl 10.0.2.57:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
```

You can verify now that no other pod apart from the dev1 namespace is able to connect to the pod in the dev2 namespace.

For this, simply create a pod in the default namespace and try to connect to the pod in the dev2 namespace.

```
kubectl run df-demo --image=nginx
pod/df-demo created
```

Let's execute the same command again but this time tyron to connect from the `df-demo` pod to the `demo-dev2` pod in the dev2 namespace.

```
root@controlplane-a5cc-270721:/# kubectl exec -it df-demo -- curl 10.0.2.57:80
curl: (28) Failed to connect to 10.0.2.57 port 80 after 130504 ms: Couldn't connect to server
command terminated with exit code 28
```

This is the correct behaviour as the NetworkPolicy created for the pods in dev2 namespace only allows connection on port 80 from the pods in dev1 namespace. All other pods in any other namespace won't be able to connect to the pods in dev2 namespace.
This is how you can create a strong security isolation.

Cleanup

```
root@controlplane-a5cc-270721:/# kubectl delete pod df-demo
pod "df-demo" deleted
root@controlplane-a5cc-270721:/# kubectl delete ns dev2
namespace "dev2" deleted
root@controlplane-a5cc-270721:/# kubectl delete ns dev1
namespace "dev1" deleted
```

Scenario 22 - RBAC - Role based access control

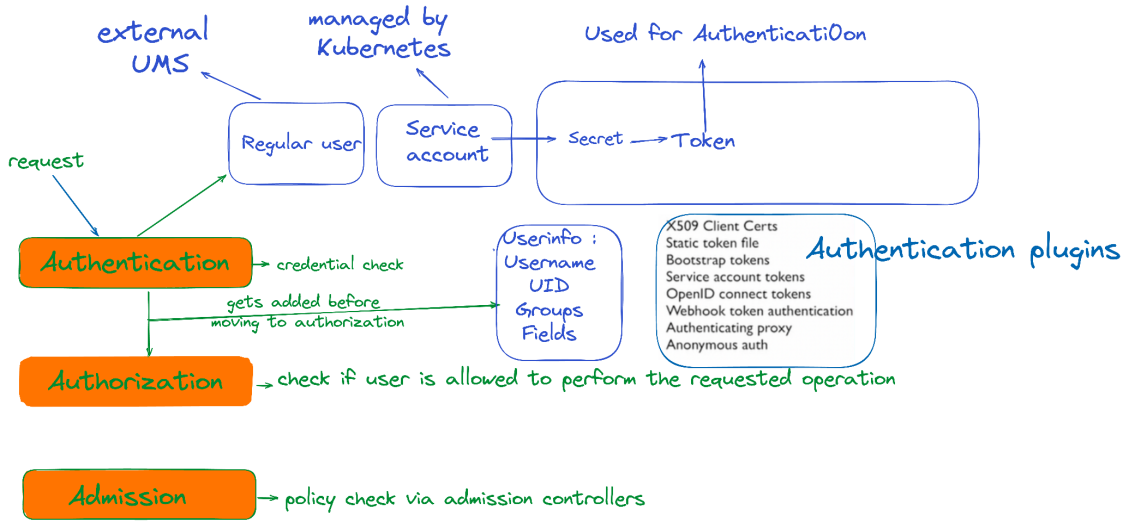
Create a service account ‘demo-sa’ in app namespace
Create a cluster role ‘my-rules’ that can only create deployments and daemonsets.
Create a service account ‘demo2-sa’ that can only create deployments in dev1 namespace

Solution 22 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster(Kind, Rancher Desktop, etc.)

In Kubernetes there is a concept of RBAC that can be used to provide access to specific resources to the service account.
How that works is that you create a service account in a particular namespace, then you create a Role with the verb(what actions can be performed like create,delete etc.) and resources(like deployments, secrets etc.), after this you create a RoleBinding where you bind the Role to a service account.

ClusterRole + RoleBinding - Can be used
Role + RoleBinding - Can be used
ClusterRole + ClusterRoleBinding - Can be used
Role + ClusterRoleBinding - WRONG



You can create the service account just using the imperative command.

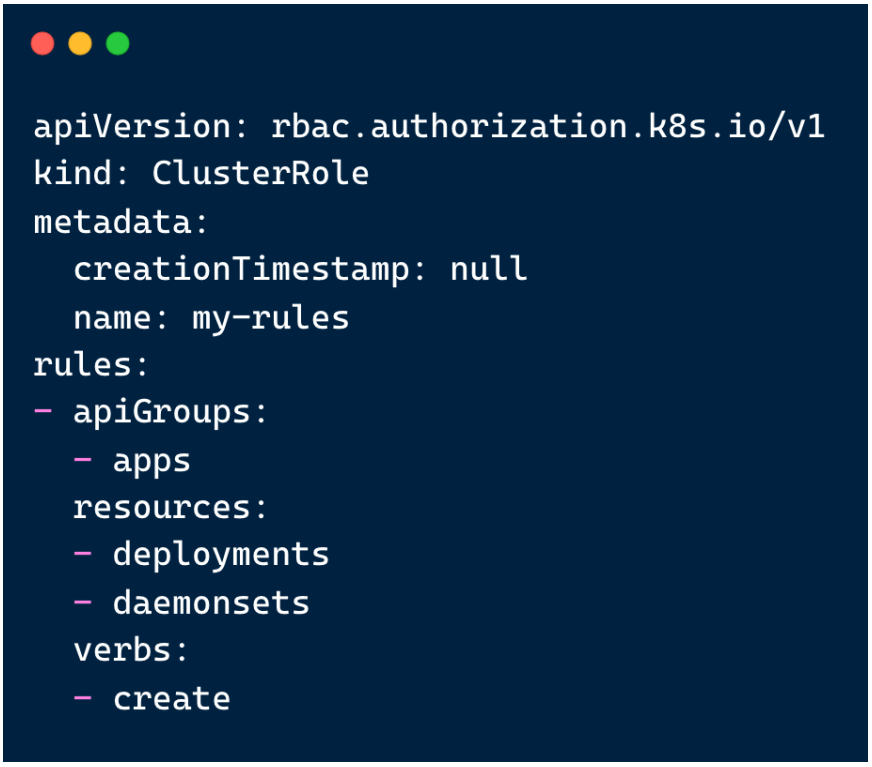
```
kubectl create sa demo-sa -n app
```

```
root@controlplane-a5cc-270721:/# kubectl create ns app
namespace/app created
root@controlplane-a5cc-270721:/# kubectl create sa demo-sa -n app
serviceaccount/demo-sa created
root@controlplane-a5cc-270721:/# kubectl get sa -n app
NAME      SECRETS   AGE
default   0         20s
demo-sa   0         8s
```

The next part of this scenario is to create a ClusterRole that only create deployment and daemonset

It can be done using this command

```
kubectl create clusterrole my-rules --verb=create --resource=deployments
--resource=daemonsets -o yaml --dry-run=client > my-rules.yaml
```



```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: null
  name: my-rules
rules:
- apiGroups:
  - apps
  resources:
  - deployments
  - daemonsets
  verbs:
  - create
```

You can see from the above yaml that you can specify rules and in those rules you can specify apiGroups, resources and verbs(what action the role should have). Post creating this you can use a ClusterRoleBinding to map it to a service account.

For applying this you can simply use `kubectl apply`.

```
kubectl apply -f my-rules.yaml
clusterrole.rbac.authorization.k8s.io/my-rules created
```

Now there can be a little complex scenario where you are asked to not only create a service account but also limit its powers. Like the last part of this scenario where you need to create a service account ‘demo2-sa’ that can only create deployments in the dev1 namespace.

For this there are multiple Kubernetes resources that need to be created.
Create the namespace

```
kubectl create ns dev1
namespace/dev1 created
```

Create Service account

```
kubectl create serviceaccount demo2-sa -n dev1
serviceaccount/demo2-sa created
```

Create Role

```
kubectl create role demo2-deployment-creator --verb=create
--resource=deployments.apps -n dev1
role.rbac.authorization.k8s.io/demo2-deployment-creator created
```

Create RoleBinding

```
kubectl create rolebinding demo2-sa-deployment-binder
--role=demo2-deployment-creator --serviceaccount=dev1:demo2-sa -n dev1
rolebinding.rbac.authorization.k8s.io/demo2-sa-deployment-binder created
```

You can see that the RoleBinding resource binds the role to a service account.

```
root@controlplane-a5cc-270721:/# kubectl create ns dev1
namespace/dev1 created
root@controlplane-a5cc-270721:/# kubectl create serviceaccount demo2-sa -n dev1
serviceaccount/demo2-sa created
root@controlplane-a5cc-270721:/# kubectl create role demo2-deployment-creator --verb=create --resource=deployments.apps -n dev1
role.rbac.authorization.k8s.io/demo2-deployment-creator created
root@controlplane-a5cc-270721:/# kubectl create rolebinding demo2-sa-deployment-binder --role=demo2-deployment-creator --serviceaccount=dev1:demo2-sa -n dev1
rolebinding.rbac.authorization.k8s.io/demo2-sa-deployment-binder created
```

Now you can use `kubectl auth can-i` to verify

```
kubectl auth can-i create deployments --namespace dev1
--as=system:serviceaccount:dev1:demo2-sa
```

```
root@controlplane-a5cc-270721:/# kubectl auth can-i create deployments --namespace dev1 --as=system:serviceaccount:dev1:demo2-sa
yes
root@controlplane-a5cc-270721:/# kubectl auth can-i create secrets --namespace dev1 --as=system:serviceaccount:dev1:demo2-sa
no
```

Let’s try to up the game and do some complex things and actually use this service account to create a secret.
You cannot do it directly so I will be showing the curl command to interact with the api directly using the service account and token.
To generate the token for a service account you can use `kubectl create token demo2-sa -n dev1`

Set the TOKEN form the output of the above command.

```
root@controlplane-a5cc-270721:/# kubectl create token demo2-sa -n dev1
eyJhbGciOiJSUzI1NiIsImtpZCI6IkVJNUJLT3ZSbkpXME4waE5PVFNNdjRNbHpsbUF6b0lDS0Ryc3BsZzB3SlEifQ.ey
JhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiXSwiZXhwIjoxNjkxNDgzOTA
xLCJpYXQiOjE2OTE0ODAzMDEsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyLmxvY2Fs
Iiwia3ViZXJuZXRlcy5pbyI6eyJuYW1lc3BhY2UiOiJkZXYxIiwic2VydmljZWFjY291bnQiOnsibmFtZSI6ImRlbW8yL
XNhIiwidWlkIjoiYWY2N2U2OGQtMTQ3NC00NzM0LTg2ZTQtNTcyZTJmZjM4NjE3In19LCJuYmYiOjE2OTE0ODAzMDEsIn
N1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDpkZXYxOmRlbW8yLXNhIn0.Jcud44ZXvHqvtrnlqQJc8hldqAB_PQdyv1a
wKZao0xFYuxp17kESz-yJXA7V9J3ELm_uoV7Tv5LqbNLHiFRGDPngXJL_XTh0PcgB6b2z49PtfUmymYog6GZBY-
ifUdSVG17okNmB2oHe1GOJ0mh_Xi2P3umpqs_eLdEe0Az1WTRoD70Hhe91wXzJYgYuC9OBF7PJyRdVaH_rNfocN8YyKjA
4Vh0bRwfFhXgRYjDoxZjAYG3Qk2T_kx4lQYbhAU7U0ashfq0rOpH-szG1rPHBaTJ_lpBwkpfbrvYTzvf7-
bQA1KOT5ZoW_0ziE3WJSAOhr713b3Z-lSKGTbnFMCLhjw

root@controlplane-a5cc-270721:/#
TOKEN="eyJhbGciOiJSUzI1NiIsImtpZCI6IkVJNUJLT3ZSbkpXME4waE5PVFNNdjRNbHpsbUF6b0lDS0Ryc3BsZzB3Sl
EifQ.eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiXSwiZXhwIjoxNjkx
NDgzOTAxLCJpYXQiOjE2OTE0ODAzMDEsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyL
mxvY2FsIiwia3ViZXJuZXRlcy5pbyI6eyJuYW1lc3BhY2UiOiJkZXYxIiwic2VydmljZWFjY291bnQiOnsibmFtZSI6Im
RlbW8yLXNhIiwidWlkIjoiYWY2N2U2OGQtMTQ3NC00NzM0LTg2ZTQtNTcyZTJmZjM4NjE3In19LCJuYmYiOjE2OTE0ODA
zMDEsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDpkZXYxOmRlbW8yLXNhIn0.Jcud44ZXvHqvtrnlqQJc8hldqAB_
PQdyv1awKZao0xFYuxp17kESz-
yJXA7V9J3ELm_uoV7Tv5LqbNLHiFRGDPngXJL_XTh0PcgB6b2z49PtfUmymYog6GZBY-
ifUdSVG17okNmB2oHe1GOJ0mh_Xi2P3umpqs_eLdEe0Az1WTRoD70Hhe91wXzJYgYuC9OBF7PJyRdVaH_rNfocN8YyKjA
4Vh0bRwfFhXgRYjDoxZjAYG3Qk2T_kx4lQYbhAU7U0ashfq0rOpH-szG1rPHBaTJ_lpBwkpfbrvYTzvf7-
bQA1KOT5ZoW_0ziE3WJSAOhr713b3Z-lSKGTbnFMCLhjw"
```

Get the APISERVER

```
APISERVER=$(kubectl config view --minify -o
jsonpath='{.clusters[0].cluster.server}')
```

Do the CURL request to create a secret

```
curl -k -X POST $APISERVER/api/v1/namespaces/dev1/secrets \
    -H "Authorization: Bearer $TOKEN" \
    -H "Content-Type: application/json" \
    -d '{
        "apiVersion": "v1",
        "kind": "Secret",
        "metadata": {
            "name": "demo-secret"
        },
        "data": {
            "key": "'$SECRET_DATA'"
        }
    }'
```

```
root@controlplane-a5cc-270721:/# curl -k -X POST $APISERVER/api/v1/namespaces/dev1/secrets \
    -H "Authorization: Bearer $TOKEN" \
    -H "Content-Type: application/json" \
    -d '{
        "apiVersion": "v1",
        "kind": "Secret",
        "metadata": {
            "name": "demo-secret"
        },
        "data": {
            "key": "'$SECRET_DATA'"
        }
    }'
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "secrets is forbidden: User \"system:serviceaccount:dev1:demo2-sa\" cannot
create resource \"secrets\" in API group \"\" in the namespace \"dev1\"",
  "reason": "Forbidden",
  "details": {
    "kind": "secrets"
  },
  "code": 403
```

You can see in the message section that it is forbidden to create a secret in the dev1 namespace using a demo2-sa service account.

```
root@controlplane-a5cc-270721:/# curl -k -X GET
$APISERVER/apis/apps/v1/namespaces/dev1/deployments \
    -H "Authorization: Bearer $TOKEN"
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "deployments.apps is forbidden: User \"system:serviceaccount:dev1:demo2-sa\"
cannot list resource \"deployments\" in API group \"apps\" in the namespace \"dev1\"",
  "reason": "Forbidden",
  "details": {
    "group": "apps",
    "kind": "deployments"
  },
  "code": 403
```

You can also test other combinations like get deployment also won't work as this is only for creating Deployment.
This is how you can try even more complex scenarios and try to create a service account, a role or cluster role and then bind them together using rolebinding or cluster role binding.

Cleanup

```
kubectl delete ns dev11
namespace "dev1" deleted
```

Scenario 23 - Scheduling -> Mark node as unschedulable

Mark worker-1 as a non schedulable node so that no workload can be scheduled on this node. Verify it by trying to create a pod and scheduling it to that node.

Solution 23 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster(Kind, Rancher Desktop, etc.)

In order to make a node not available for the workloads you have to make it `SchedulingDisabled` and the command for that is `cordon`.

```
kubectl get nodes
NAME                        STATUS   ROLES           AGE     VERSION
controlplane-a5cc-270721    Ready    control-plane   4d15h   v1.27.2
worker-1-937f-617615        Ready    <none>          4d12h   v1.27.2
worker-2-e26c-617615        Ready    <none>          4d12h   v1.27.1
```

All nodes are ready, now cordon the worker-1 node.

```
kubectl cordon worker-1-937f-617615
node/worker-1-937f-617615 cordoned
```

```
kubectl get nodes
NAME                        STATUS                     ROLES           AGE     VERSION
controlplane-a5cc-270721    Ready                      control-plane   4d15h   v1.27.2
worker-1-937f-617615        Ready,SchedulingDisabled   <none>          4d12h   v1.27.2
worker-2-e26c-617615        Ready                      <none>          4d12h   v1.27.1
```

Let's try to verify that by trying to create a pod and schedule it on worker-1 node.

```
kubectl run nginx --image=nginx --dry-run=client -oyaml > pod.yaml
```

Edit this manifest and add nodeSelector to schedule on worker-1 node.

```
spec:
  nodeSelector:
    CKA: "true"
```

Apply the pod and you will see that the pod will be in pending state. You can also do

`kubectl describe pod` and see the reason.

```
root@controlplane-a5cc-270721:/# vi pod.yaml
root@controlplane-a5cc-270721:/# kubectl apply -f pod.yaml
pod/nginx created
root@controlplane-a5cc-270721:/# kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   0/1     Pending   0          21s
```

`kubectl describe pod nginx`: you can see the events clearly states that 1 node was unschedulable and other 2 do not match the selector used.

```
spec:
  Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  67s   default-scheduler  0/3 nodes are available: 1 node(s) were
unschedulable, 2 node(s) didn't match Pod's node affinity/selector. preemption: 0/3 nodes are
available: 3 Preemption is not helpful for scheduling..nodeSelector:
    CKA: "true"
```

You can uncordon the node and mark it schedulable again

```
kubectl uncordon worker-1-937f-617615
node/worker-1-937f-617615 uncordoned
```

As soon as you do this, the pod will get scheduled on this node and it will go into the running state.

```
kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          8m52s
```

Cleanup

```
kubectl delete pod nginx
pod "nginx" deleted
```

Scenario 24 - Pod with specific Service Account

Create a pod demo using the image nginx and service account demo-sa in dev1 namespace.

Solution 24 -

In this scenario, you can use the cluster that you created in the Scenario 1
Or any other Kubernetes cluster(Kind, Rancher Desktop, etc.)

Create the namespace

```
kubectl create ns dev1
namespace/dev1 created
```

Create the service account demo-sa

```
kubectl create serviceaccount demo-sa -n dev1
serviceaccount/demo-sa created
```

Create the pod using this service account

```
kubectl run demo --image=nginx -n dev1 --dry-run=client -o json
--overrides='{"spec": {"serviceAccountName": "demo-sa"}}' | kubectl
apply -f -
pod/demo created
```

You can do it via the YAML file or using the imperative way using the `--overrides` flag where I have provided the `serviceAccountName` information.

```
root@controlplane-a5cc-270721:/# kubectl get pods -n dev1
NAME   READY   STATUS    RESTARTS   AGE
demo   1/1     Running   0          31s
root@controlplane-a5cc-270721:/# kubectl get pod demo -n dev1 -oyaml | grep serviceAccount
      {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":
{},"creationTimestamp":null,"labels":{"run":"demo"},"name":"demo","namespace":"dev1"},"spec":
{"containers":[{"image":"nginx","name":"demo","resources":
{}}],"dnsPolicy":"ClusterFirst","restartPolicy":"Always","serviceAccountName":"demo-
sa"},"status":{}}
  serviceAccount: demo-sa
  serviceAccountName: demo-sa
      - serviceAccountToken:
```

Cleanup

```
kubectl delete ns dev1
namespace "dev1" deleted
```

Scenario 25 - Resize Volumes

Create a persistent volume claim saiyam with the default storage class and 1Gi capacity. Mount it to a pod demo using image nginx at `/usr/share/nginx/html/index.html`, resize the pvc to 2Gi and record that event.

Solution 25 -

Creation of PVC is simple as that have been already discussed in Scenario 17

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: saiyam
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

```
root@controlplane-a5cc-270721:/# vi pvc.yaml
root@controlplane-a5cc-270721:/# kubectl apply -f pvc.yaml
persistentvolumeclaim/saiyam created
root@controlplane-a5cc-270721:/# kubectl get pvc
NAME     STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
saiyam   Bound    pvc-6e96ee57-c5f9-4438-83a2-31f9b313b0ed   1Gi        RWO
longhorn       32s
root@controlplane-a5cc-270721:/# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
  CLAIM            STORAGECLASS   REASON   AGE
pvc-6e96ee57-c5f9-4438-83a2-31f9b313b0ed   1Gi        RWO            Delete           Bound
default/saiyam   longhorn               5s
```

Create the pod

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  volumes:
    - name: pv-saiyam
      persistentVolumeClaim:
        claimName: saiyam
  containers:
```

```
    - name: demo
      image: nginx
      ports:
        - containerPort: 80
          name: "http"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: pv-saiyam
```

Apply the pvc and pod

```
kubectl apply -f pvc.yaml
kubectl apply -f pod.yaml
```

```
root@controlplane-a5cc-270721:/# vi pvc.yaml
root@controlplane-a5cc-270721:/# kubectl apply -f pvc.yaml
persistentvolumeclaim/saiyam created
root@controlplane-a5cc-270721:/# kubectl get pvc
NAME     STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
saiyam   Bound    pvc-6e96ee57-c5f9-4438-83a2-31f9b313b0ed   1Gi        RWO
longhorn       32s
root@controlplane-a5cc-270721:/# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
  CLAIM            STORAGECLASS   REASON   AGE
pvc-6e96ee57-c5f9-4438-83a2-31f9b313b0ed   1Gi        RWO            Delete           Bound
default/saiyam   longhorn               5s
```

Before running the command to update the storage, make sure that allowVolumeExpansion: true is added to the storage class, Since I have used Longhorn through so this is already present in the longhorn storage class.

Update the pvc with 2Gi storage

```
kubectl patch pvc my-pvc -p '{"spec": {"resources": {"requests":
{"storage": "2Gi"}}}}' --record
```

```
kubectl patch pvc saiyam -p '{"spec": {"resources": {"requests": {"storage": "2Gi"}}}}' --
record
Flag --record has been deprecated, --record will be removed in the future
persistentvolumeclaim/saiyam patched
kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
  CLAIM            STORAGECLASS   REASON   AGE
pvc-82a9b6ca-1338-484c-b863-be59e30c9b3c   2Gi        RWO            Delete           Bound
default/saiyam   longhorn               3h29m
```

You can check the resize triggered events when you describe the pvc.

```
  Type     Reason                      Age    From                                Message
  ----     ------                      ----   ----                                -------
  Warning  ExternalExpanding           7m51s  volume_expand                       waiting
for an external controller to expand this PVC
  Normal   Resizing                    7m50s  external-resizer driver.longhorn.io  External
resizer is resizing volume pvc-82a9b6ca-1338-484c-b863-be59e30c9b3c
  Normal   FileSystemResizeRequired    7m44s  external-resizer driver.longhorn.io  Require
file system resize of volume on node
  Normal   FileSystemResizeSuccessful  7m12s  kubelet
MountVolume.NodeExpandVolume succeeded for volume "pvc-82a9b6ca-1338-484c-b863-be59e30c9b3c"
worker-2-e26c-8047c4
```

You may have noticed that I have used the record command to record this change(although this will be deprecated soon).
This recorded change you can see when you get the pvc with `-oyaml`

```
root@controlplane-a5cc-43e751:~# kubectl get pvc -oyaml
apiVersion: v1
items:
- apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"v1","kind":"PersistentVolumeClaim","metadata":{"annotations":
{},"name":"saiyam","namespace":"default"},"spec":{"accessModes":
["ReadWriteOnce"],"resources":{"requests":{"storage":"1Gi"}}}}
      kubernetes.io/change-cause: 'kubectl patch pvc saiyam --patch={"spec": {"resources":
        {"requests": {"storage": "2Gi"}}}} --record=true'
```

You can see the annotations that got added to the pvc.

This way you can resize or expand volume.

Cleanup

```
root@controlplane-a5cc-43e751:~# kubectl delete pod --all
pod "demo" deleted
root@controlplane-a5cc-43e751:~# kubectl delete pvc --all
persistentvolumeclaim "saiyam" deleted
```

Scenario 26 - ConfigMap and Secrets

Create a configmap demo with key=value (colour=green, name=saiyam, exam=cka) and mount that onto the pods to be used as environment variables.

Create a secret cka-demo with key=value (password=iwillclearcka) and mount that to a pod to use it as a DATABASE_PASSWORD variable.

Solution 26 -

A Config map is used to store non-confidential data like the hostnames etc or the environment variables that your applications might need. Data is stored in key/value pairs and the max size for a config map is 1MB. As per the Kubernetes documentation configmap can be used in 4 different ways:

- Inside a container command and args
- Environment variables for a container
- Add a file in read-only volume, for the application to read
- Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

In order to create the configmap you can simply use the imperative command

```
kubectl create configmap demo --from-literal=colour=green
--from-literal=name=saiyam --from-literal=exam=cka
```

Once created, you can map it within the pod via the envFrom

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
  - name: demo-container
    image: nginx
    envFrom:
    - configMapRef:
        name: demo
```

In order to check, you can exec into the pod and check the env.

```
kubectl exec demo-pod -- env
```

```
root@controlplane-a5cc-43e751:~# kubectl create configmap demo --from-literal=colour=green --
from-literal=name=saiyam --from-literal=exam=cka
configmap/demo created
root@controlplane-a5cc-43e751:~# vi pod.yaml
root@controlplane-a5cc-43e751:~# kubectl create -f pod.yaml
pod/demo-pod created
root@controlplane-a5cc-43e751:~# kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
demo-pod   1/1     Running   0          5s
root@controlplane-a5cc-43e751:~# kubectl exec demo-pod -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=demo-pod
NGINX_VERSION=1.25.1
NJS_VERSION=0.7.12
PKG_RELEASE=1~bookworm
exam=cka
name=saiyam
colour=green
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
HOME=/root
```

Let's go a step further and try to use another way to get key/value pairs inside the pods. You can mount configmaps as a volume and all the key/values will be a separate file under the specified mount path.

For this create the config map -> use the same one as created previously.
Delete the previous pod

```
kubectl delete pod demo-pod
pod "demo-pod" deleted
```

Create a new pod with following manifest

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
    - name: demo-container
      image: nginx
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
```

```
  volumes:
    - name: config-volume
      configMap:
        name: demo
```

In above, you can see that the volume section has `configMap`.

In order to check if the files are created, use the exec command again on the mountPath inside the container.

```
kubectl exec -it demo-pod -- ls /etc/config
colour      exam  name
```

```
kubectl exec -it demo-pod -- cat /etc/config/exam
cka
```

So the Key name becomes the file name and those can be used as variables or however required by your application.
You can also create an immutable configmap by setting immutable: true in the data section of the configMap.
Let's create a bit more complex config map. Create a file `cm.yaml` with below contents:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo2
  namespace: default
data:
  #single key value
  colour: green
  exam: cka
  name: saiyam
    # multiple values when you mount as volume
  k8s.properties: |
    exam.names=kcna,cka,ckad,cks
    attendee.attempts=2
```

Mount it as a volume and see the file contents

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod2
spec:
  containers:
    - name: demo-container
      image: nginx
```

```
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: demo2
```

```
root@controlplane-a5cc-43e751:~# kubectl exec -it demo-pod2 -- ls /etc/config
colour  exam  k8s.properties  name
root@controlplane-a5cc-43e751:~# kubectl exec -it demo-pod2 -- cat /etc/config/k8s.properties
exam.names=kcna,cka,ckad,cks
attendee.attempts=2
```

The properties one is interesting here as in a single file you will be able to have multiple pairs.

These are various ways of creating and mounting a configMap inside the pod.

Secrets
Moving onto the secrets, secrets in Kubernetes are a way to store sensitive information. Although the Kubernetes secrets are not so secure and there are various projects like secret store CSI driver, external Secretes operator etc that can be used for managing Kubernetes secrets. But for this scenario, let's create the Kubernetes secret for the database password.

Meanwhile if you would like to know about the projects mentioned, I have created detailed videos on that including examples.

Links -> Sealed Secrets , External secretes Operator, Secret Store CSI Driver

You can create the K8s secret using the imperative command.

```
kubectl create secret generic cka-demo
--from-literal=password=iwillclearcka
secret/cka-demo created
```

```
root@controlplane-a5cc-270721:~# kubectl create secret generic cka-demo --from-
literal=password=iwillclearcka
secret/cka-demo created

root@controlplane-a5cc-270721:~# kubectl get secret cka-demo -oyaml
apiVersion: v1
data:
  password: aXdpbGxjbGVhcmNrYQ==
kind: Secret
metadata:
  creationTimestamp: "2023-08-09T10:09:51Z"
  name: cka-demo
  namespace: default
  resourceVersion: "1460292"
  uid: fcff40b1-4bf7-4948-a5de-0b56c6636216
type: Opaque

root@controlplane-a5cc-270721:~# echo "aXdpbGxjbGVhcmNrYQ==" | base64 -d
iwillclearcka
```

Create the `pod.yaml` file with below contents to use that secret created.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
  - name: demo-container
    image: busybox
    command:
      - sleep
      - "3600"
    env:
      - name: DATABASE_PASSWORD
        valueFrom:
          secretKeyRef:
            name: cka-demo
            key: password
```

You can check it with `kubectl exec -it demo-pod -- env | grep DATABASE_PASSWORD`

```
root@controlplane-a5cc-270721:~# kubectl create -f pod.yaml
pod/demo-pod created
root@controlplane-a5cc-270721:~# vi pod.yaml ^C
root@controlplane-a5cc-270721:~# kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
demo-pod   1/1     Running   0          5s
root@controlplane-a5cc-270721:~# kubectl exec -it demo-pod -- env | grep DATABASE_PASSWORD
```

Some other type of secrets that can be created as per Kubernetes documentation are below:

Built-in Type	Usage
`Opaque`	arbitrary user-defined data
`kubernetes.io/service-account-token`	ServiceAccount token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file
`kubernetes.io/basic-auth`	credentials for basic authentication
`kubernetes.io/ssh-auth`	credentials for SSH authentication
`kubernetes.io/tls`	data for a TLS client or server
`bootstrap.kubernetes.io/token`	bootstrap token data

This is how you can create configMap and secrets in Kubernetes and use them inside a pod.

Cleanup

```
kubectl delete pod --all
```

Go for it

The more you practise, the more efficient you will become. Clearing the exam is not the main objective here, the main objective is to understand all the concepts and scenarios so that when you are actually managing the Kubernetes infra part, you know what is happening inside of the system and how to troubleshoot it.

Again, the learning resources are there on my Kubesimplify Youtube channel where you can go through the Kubernetes workshop, taints and tolerations video, other Kubernetes secrets videos and much more wrt cloud native.

Once you understand the concepts and all the scenarios here and are able to do it quickly in less time. You should be ready for the exam.

Wish you all the best, thank you for purchasing this book and supporting my work.
Do share it with your network and once you clear CKA feel free to tag me in your achievement.

I am on all major platforms with saiyampathak.