Core concepts

k8s cluster has master nodes and worker nodes. Master node manages the worker nodes, comprises of api-server, scheduler, node controller, replication controller, etcd DB. Worker nodes have kubelet, kube-proxy, container runtime.

- **docker vs containerd:**

k8s was initially created to orchestrate docker but later CRI (container runtime Interface) came into picture to support other container engines like rkt, k8s developed a Open container initiative (OCI) standards to support all runtimes.
docker's had other features like api, build, image repo, volumes etc., and dockers container runtime is - containerd which is OCI standard met.
nerdctl is the CLI for containerd.

- **ETCD**

etcd is a distributed, reliable key-value db. we can use "etcdctl put key1 value1" to insert a entry and "etcdctl get key1" to retrieve value. etcd comes with etcdctl cli tool and default api version as 2 . etcd runs on port 2379.
To set the right version of API set the environment variable ETCDCTL_API, command:
`export ETCDCTL_API=3` .

- **Kube api server**

when we execute a kubectl command to get info, we are actually interacting with kube-apiserver and which in turns gets the data from etcd DB. we can also create objects, get info using api calls.
kube api server is the central point which performs : User authentication, request validation, talking to etcd cluster/ scheduler/ kubelet.

- **kube controller manager**

There are multiple controllers for the most objects in k8s. Node controller - watches nodes every 5s and if unhealthy, marks it and gives a grace period of 40s and if not coming back, it'll remove the pods deployed on them to other node in 5mins.
Replication controller - makes sure the desired no.of replicasets are maintained, if any of the pod is removed, it creates new one.

- **scheduler**

It chooses the node based on the pod requirement (best pick out of available nodes). However, Kubelet is the one to deploy a pod on the node chosen by scheduler.

- **Kubelet**

Kubelet on worker nodes: Register a node under cluster, creates PODs as instructed by scheduler, monitors PODs and node; updates their status to kube-api-server.

NOTE: kudeadm doesnot deploy kubelet by default

- **Kubeproxy**

pods in a cluster can communicate with each other using their IPs as there will be a pod network by default. But IP communication is not suggested as the Pod IPs are dynamic. Service communication is best. Service is a virtual object in k8s and it gets an IP. Kubeproxy maintains routing table on each node mapping service IP to the backend Pod IP.

- kubectl scale replicaset new-rs --replicas=5

scales up replicas and update the background replica set file.

kubectl edit replicaset new-rs

can be used to edit the replica set file in runtime, this also scales replicas automatically.

Services:

- **Node Port:** The web application/ any application intended to be access by users, we can expose it as a port on the node. (30000-32767 nodeport range). Based on the selector filed, it spans over the pods within a single node/ spread over diff nodes in cluster as well. It also acts as loadbalancer, routes the traffic from NodeIP:Nodeport to one of the pods.

spec:

type: NodePort

ports:

- nodePort: 30002

  port: 80 (only mandatory field)

  targetPort: 80

selector:

app: frontend

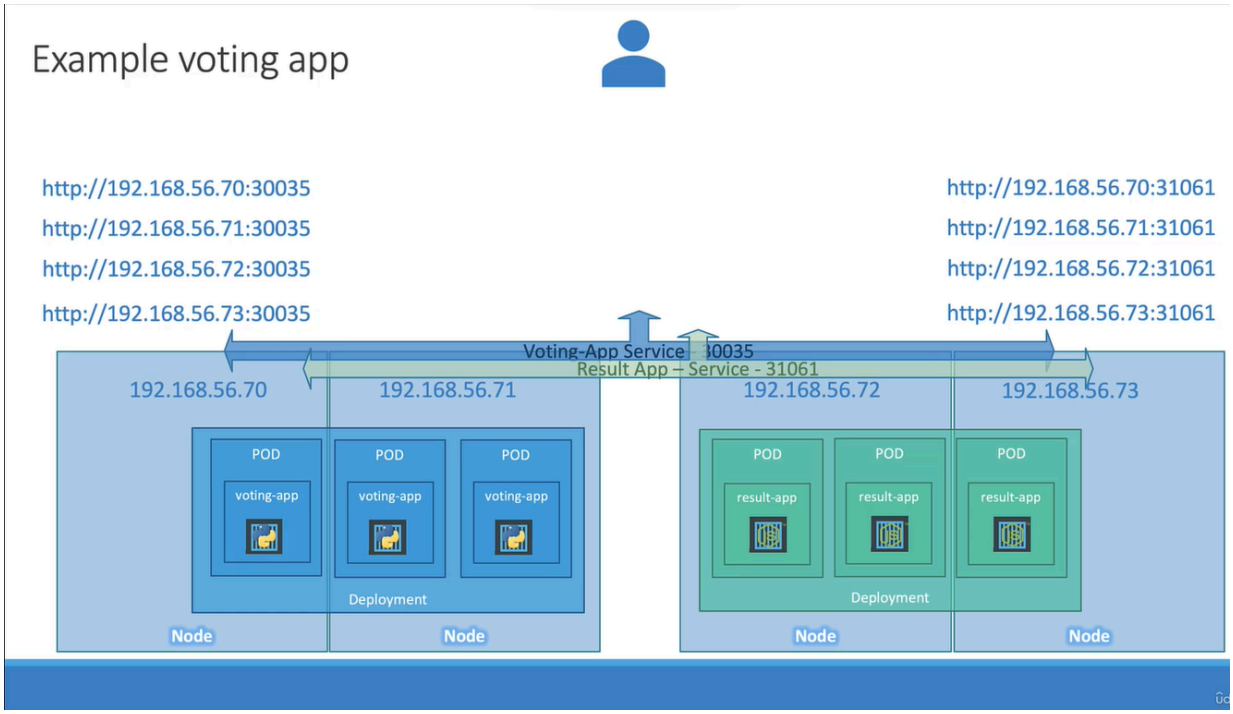
type: webapp

--> target port is the port on pod. If not specified, port=targetport & nodeport will be assgined on availability.

Cluster Ip: For one set of pods (serving one application-frontend) to communicate with other application pods, rather than using IP's which are dynamic, we can create service-cluster IP which binds all the pods under it and adjusts to the scaling/ re-creation of pods.

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  type: ClusterIP (default service type)
  ports:
   -  targetPort: 80
      port: 80
  selector:
     app: my-app
     type: backend
```

- so frontend pods can now talk to backend set of pods via the service name- "backend-service" or the created clusterIp.



NOTE: though voting-app pods are created only on 2 nodes/ 4 nodes, those will be accessible via IP's of other 2 nodes. (node ip: Port)

LoadBalancer: The way of creating object is same as nodeport, except type: LoadBalancer, the native cloud provider will create loadbalancer. (to be continued.,)

Namespaces:
Are virtual isolation. We can create resource quotas to a namespace.
kubectl create namespace dev --> command line/ we can also use file to create namespace object.
kubectl get pods -n dev
kubectl get pods --all-namespaces or kubectl get pods -A

application (pods) in on namespace can access its service using its name alone, but to access service in another namespace use:
servicename.namespace.svc.cluster.local
--> switch namespace:
kubectl config set-context $(kubectl config current-context) --namespace=dev

few of imperative commands:

- kubectl run nginx --image=nginx
- kubectl run redis -l tier=db --image=redis:alpine **or**

- kubectl run redis --labels="tier=db,app=redis" --image=redis:alpine
- kubectl run httpd --image=httpd:alpine --port=80 --expose (pod and service are created)
- kubectl run custom-nginx --image=nginx --port=8080 (service doesn't get created, pod just runs on port 8080)
- kubectl create deployment redis --image=redis
- kubectl create deployment redis-deploy --image=redis --replicas=2 --namespace=dev-ns
- kubectl expose deployment redis --port 80
- kubectl expose pod redis --name=redis-service --port=6379
- kubectl scale replicaset ngnix --replicas=5
- kubectl edit deployment redis
- kubectl set image deployment frontend nginx=nginx:1.5 (nginx is the container name)
- kubectl create -f deploy.yaml
- kubectl edit -f deployment.yaml (to edit the file)
- kubectl replace --force -f nginx.yaml (to get the changes effective)
- kubectl delete -f deploy.yaml

apply command creates if object is not existing, if file is changed for an existing object, it updates accordingly.

- kubectl create deployment nginx --image=nginx --dry-run=client -o yaml > nginx-deployment.yaml
  It will generate a yaml file, can be used to edit and apply the changes.

Scheduling

- scheduler in kubernetes usually run as a pod, if scheduler is not present, we have to manually schedule in pod in one of the node.

spec:
nodeName: node01
containers:
- name: nginx
  image: nginx
  ports:
    - containerPort: 80

- kubectl replace --force -f nginx.yaml (will delete and recreate pod in specified node)
- kubectl get pods --selector env=dev (to gets pods having the label) / k get pods -l env=dev
- kubectl get pods --selector env=dev,app=frontend (to gets pods having both the labels)

we cannot assign a node to already created pod. In this case, we will have to create a pod-binding object and curl request to the binding api.
ex:
apiVersion: v1
kind: Binding
metadata:
  name: nginx
spec:
  target:
    apiVersion: v1
    kind: Node
    name: node2
curl request ex:
curl --header "Content-Type:application/json" --request POST --data '{"apiVersion":"v1", "kind": "Binding“ …. }' http://$SERVER/api/v1/namespaces/default/pods/$PODNAME/binding/
(we can also update the nodeName to yaml and use --> kubectl replace --force -f nginx.yaml for both pending and running pods but
NOTE that for a running pod, if we updated the nodename; it deletes and recreates pod but doesn't move from one node to other)

Taints and Tolerations:

- Taints are applied to nodes and only the pods that tolerate that setting, will be scheduled in a "Noschedule" taint effect
- ex: kubectl taint nodes <nodename> key=value:tainteffect

taint effect --> can be NoSchedule/ PreferNoSchedule/ NoExecute
NoSchedule -- will not schedule any pods if taint not satisfied.
PreferNoSchedule -- will try not to
NoExecute -- new pods will not be scheduled if taint not satisfied and even evicts any unsatisfied existing pods.

- Applying tolerations to Pods:
  - spec:
    containers:
    - name: nginx
      image: nginx
    tolerations:
    - key: "app"
      operator: "Equal"
      value: "blue"
      effect: "NoSchedule"

Interesting Fact: Taints and tolerations can only restrict - any pod which doesn't satisfy the taint from scheduling on that node; which doesn't mean that any pod which has the required tolerations should not be scheduled on other plain nodes. kubectl taint nodes node1 key1=value1:NoSchedule- (to remove taint from the node)

Node Selectors:

label the node u want with key=value "kubectl label node node01 size=Large" and in the pod-def file, add the field under spec section:

```
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    size: Large
```

- But if u want a more advanced restriction to schedule pod on any medium or large / any node other than small, this wont help

Node Affinity :

- Node: set the required key and value ex- kubectl label node node01 size=Large / size=Medium/ size=Small (k label nodes also working)
- Pod:

```
spec:
affinity:
nodeAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
nodeSelectorTerms:
- matchExpressions:
- key: size
operator: In
values:
- Large
- Medium
```

- node affinity types: requiredDuringSchedulingIgnoredDuringExecution
  prefferedDuringSchedulingIgnoredDuringExecution

Resource requests and Limits:

- we can set requests or limits for a container

if none of requests/ limits set, a container can use any no.of cpu/ memory.
if limits only set, requests=limits
if requests and limits both set, it works but if a container isn't using much and other one needs more than its limit, it gets starved though resources are free.
if requests set, limits not -> this is best way.

```
spec:
containers:
- name: nginx
image: nginx
resources:
requests:
cpu: 2
memory: 2Gi
limits:
cpu: 4
memory: 3Gi
```

- resource -quota helps to set total overall count for requests and limits for a namespace level

```
apiVersion: v1
kind: ResourceQuota
```

```
metadata:
name: my-quota
namespace: default
spec:
hard:
requests.cpu: 4
requests.memory: 4Gi
limits.cpu: 8
limits.memory: 10Gi
```

LimitRange: If quota is enabled in a namespace for compute resources like `cpu` and `memory`, users must specify requests or limits for those values; otherwise, the quota system may reject pod creation. Hint: Use the `LimitRanger` admission controller to force defaults for pods that make no compute resource requirements.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
  - default: # this section defines default limits
      cpu: 500m
    defaultRequest: # this section defines default requests
      cpu: 500m
    max: # max and min define the limit range
      cpu: "1"
    min:
      cpu: 100m
    type: Container
```

Daemon Sets:

- This allows to create a pod on every node. If a node added/ deleted the respective change is done.

yaml file for this is ~ to ReplicaSet/ Deployment object file except that "kind: DaemonSet" and doesn't have replicas.

- Use cases: kube-proxy/ logging/ monitoring

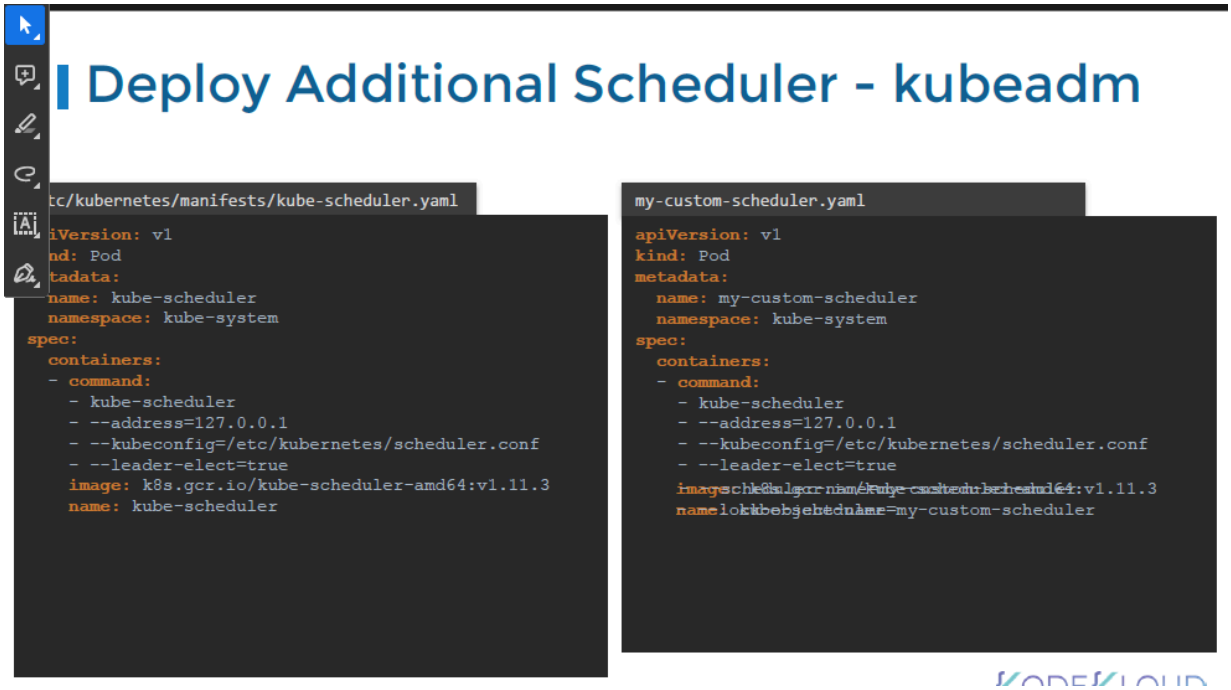
StaticPods:

If there is no master node to manage the worker nodes/ just a single worker node running standalone without master components , we can use static pods.

- Kubelet and container runtime are the available components on a worker node. Kubelet only knows deploying pods. usually, the pod yaml files placed in the /etc/kubernetes/manifests, gets picked by the kubelet.
- For creating/ deleting any pod place/ remove yaml files there.
- use docker ps / crictl ps to list pods. (as kubectl doesn't work if apiserver is not present)
- In general, if we have working cluster structure, we can use above method to create static pods. These pods metadata will be mirrored in etcd DB. so whenever we run kubectl get pods, static pods also appear in list ("-nodename" gets appended as suffix for pods). However, we cannot delete static pods using kubectl command.

Static PODs	DaemonSets
Created by the Kubelet	Created by Kube-API server (DaemonSet Controller)
Deploy Control Plane components as Static Pods	Deploy Monitoring Agents, Logging Agents on nodes
Ignored by the Kube-Scheduler	

Multiple schedulers: We can create scheduler with our custom requirements as a pod and use this scheduler to schedule pods.



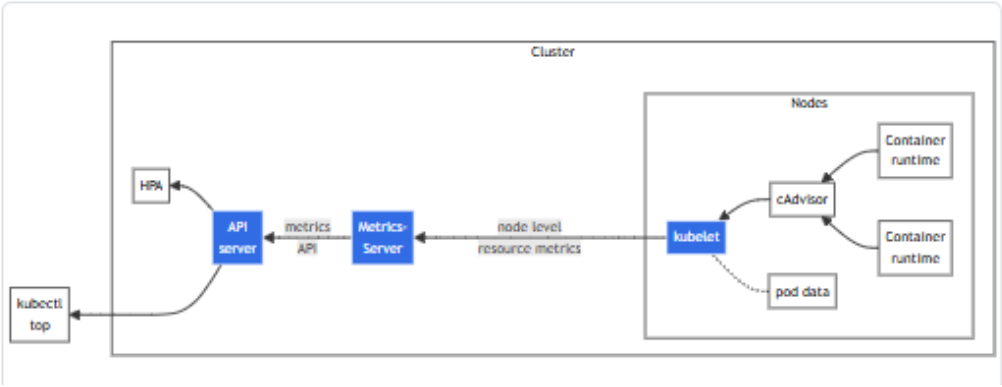
--> leader-elect is for the HA setup where there will be multiple masters and we'll have to make sure only one scheduler out of all the masters is active to avoid discrepancy.
--> In the actual pod def file, add schedulerName: my-custom-scheduler under spec: --> how the kubeconfig is attached as a volume ? --> create a configmap.yaml with the required details and in the scheduler yaml file which has serviceaccount, cluster role, cluster role bindings, deployment and even the configmap objects. And thenin the deployment object use configmap as volume (This part is uncovered as of now).
--> k get events -o wide : can be used to know which scheduler worked in scheduling the pod.
Scheduling of the pod is done in various steps : priority --> filtering --> scoring ---> binding

- you can create a PriorityClass object and can define it under the spec section of the pod yaml file. (higher the num, higher the priority), pods are arranged in queue based on this if priority is defined.
- For a pod, scheduler filters out the nodes that the pods cannot be scheduled based on the taints/ labels/ affinity/ Unschedulable : true for a node/ nodeName defined on the pod yaml etc., (These are achieved via various scheduler plugins)
- For the filtered ones, it next ranks the score based on various conditions like - Resources fit/ if image is already existing on node etc.,
- and the final chosen one gets pod bidden to it.

All the specifications here are achieved by various plugins bound via extensions. We can also have our own plugins.

LOGGING AND MONITORING :

For Kubernetes, the *Metrics API* offers a basic set of metrics to support automatic scaling and similar use cases. This API makes information available about resource usage for node and pod, including metrics for CPU and memory. If you deploy the Metrics API into your cluster, clients of the Kubernetes API can then query for this information, and you can use Kubernetes' access control mechanisms to manage permissions to do so.
we can use kubectl top node / kubectl top pods to get the metrics of cpu and memory.
The HorizontalPodAutoscaler (HPA) and VerticalPodAutoscaler (VPA) use data from the metrics API to adjust workload replicas and resources to meet customer demand.



worker node have CAdvisor which collects data and send to kubelet.

- metrics-server: Cluster addon component that collects and aggregates resource metrics pulled from each kubelet. The API server serves Metrics API for use by HPA, VPA, and by the `kubectl top` command. Metrics Server is a reference implementation of the Metrics API.
- It has only In-memory and doesn't store on disk data so cannot have any historical data, prometheus/ data dog has such provision.
- for minikube : minikube addons enable metric-server
- refer- https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/
- Managing application logs: To view logs of pod, use kubectl logs -f <pod-name> (for single container pod; -f for live logs)
- kubectl logs -f <podname> <container-name> --> floating logs incase of multiple containers in a pod.

APPLICATION LIFECYCLE MANAGEMENT

ROLLING UPDATES AND ROLLBACKS:

- When ever we create a deployment, a rollout will be created with a revision number. And when changes are applied, new revision of pods will be created.

- There are ReCreate (all pods brought down and then new ones created)/ RollingUpdate (DEFAULT - one by one pod brought down and created without app downtime) strategies.
- kubectl create -f deployment.yaml
- kubectl get deployments
- kubectl apply -f deployment.yaml / kubectl set image deployment/myapp-deployment frontendcontainer=nginx:1.31 (in the later method, actual yaml file doesn't get updated but updates in the in-memory file)
- kubectl rollout status deployment/myapp-deployment
- kubectl rollout history deployment/myapp-deployment
- kubectl rollout undo deployment/myapp-deployment

NOTE: suppose 1--> blue, 2--> green 3--> red colors of applications are changed in 3 revisions, if we do rollout undo in 3rd revision to 2nd one, 2nd revision disappears from list and 4th gets created as below:

k rollout history deployment frontend
deployment.apps/frontend
REVISION CHANGE-CAUSE
1 <none>
3 <none>
4 <none>

CONFIGURING APPLICATIONS:

- **In docker**: whenever a container is run, the container exists once the command is fulfilled/ in other words, till the application is live.

ex: "docker run ubuntu" will deploy an ubuntu container and exit as there is no terminal/command to keep it alive.

- docker run ubuntu sleep 50 (way to sleep 50 but this should be passed in command line always)
- FROM ubuntu
  CMD ["sleep", "10"]

if u build image "docker build -t ubuntu-sleeper . "(say ubuntu-sleeper image)from above and run container, it will sleep for 10 every time and exit.

- docker run ubuntu-sleeper sleep 50 (will override the command specified in docker run and sleeps for 50)
- FROM ubuntu
  ENTRYPOINT ["sleep"]

--> docker run ubuntu-sleeper 100 (above way, we can pass parameter to the entrypoint but it has no default value)

- FROM ubuntu

ENTRYPOINT ["sleep"]
CMD ["10"]

--> docker run ubuntu-sleeper 150 (we can pass value to over ride default one; if not, default gets executed)

- docker run --entrypoint sleep2.0 ubuntu-sleeper 101

--> to really override the entrypoint command, we have to pass in command-line.

In kubernetes, entrypoint ~ command and cmd ~ args in yaml file.

- If any of the command / args aren't provided in file, the defaults of the image gets executed.
  command is provided to override the entry point default value.
- we can specify command and args together like an array :

1. command: ["sleep", "100"]
2. command:
   - "sleep"
   - "200"
3. command: ["sleep"]
   args: ["200"]

Start the nginx pod using the default command, but use custom arguments (arg1 .. argN) for that command

kubectl run nginx --image=nginx -- <arg1> <arg2> ... <argN>

Start the nginx pod using a different command and custom arguments

kubectl run nginx --image=nginx --command -- <cmd> <arg1> ... <argN>

ENVIRONMENT VARIABLES IN A POD:

- For docker: docker run -e APP_COLOR=blue simple-webapp-color
- For k8s: under container spec:

```
env:
 - name: app_color
   value: blue
 - name: app_type
   value: frontend
```

```
env:
 - name: PLAYER_INITIAL_LIVES
   valueFrom:
     configMapKeyRef:
       name: game_demo
       key: player_initial_lives
   valueFrom:
     secretKeyRef:
       name:
       key:
```

Creating configmaps:

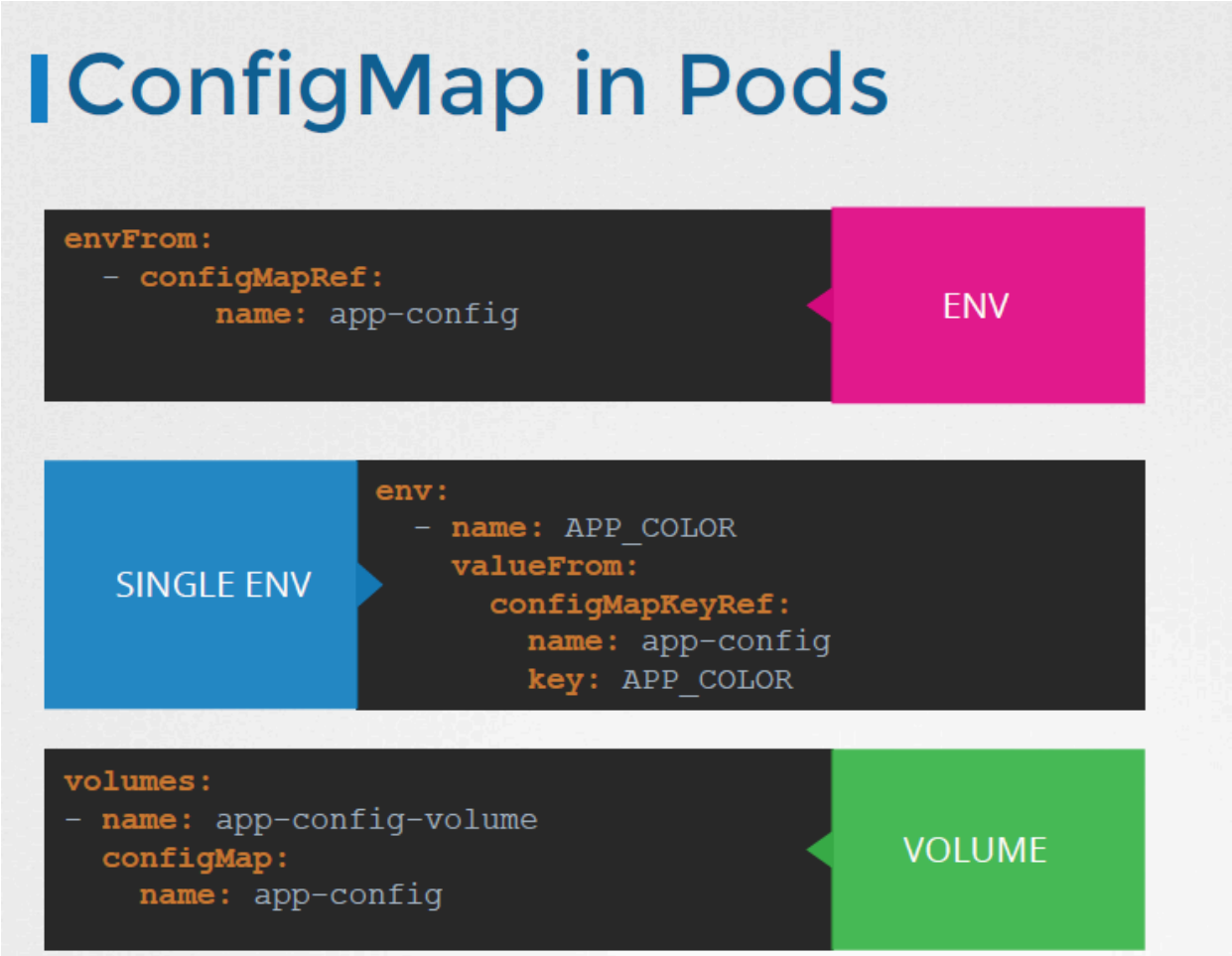
- kubectl create configmap app-config --from-literal <key1>=<value1> --from-literal <key2>=<value2>
- kubectl create configmap app-config --from-file app-config.properties

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLOR: blue
  APP_TYPE: frontend
```

--> Now to attach this configmap to the pod, under container,

```
envFrom:
- configMapRef:
    name: app-config (this is the name of configmap created)
```

- other ways:




```
env:
 - key: app-color
   value: pink
 - key: app-color
   valueFrom:
    configMapKeyRef:
      name: app-config
      key: APP_COLOR
```

--> In cases like above, the second/ the latest value will be reflected to the env variable.

Creating Secrets:

- kubectl create secret generic <secret-name> --from-literal <key>=<value> / kubectl create secret generic <secret-name> --from-literal=<key>=<value>
- kubectl create secret generic app-secret --from-file=app_secret.properties

In declarative way, we **should encode the values (echo "password" | base64)and store in the file. Though in imperative way as well, the secrets get base64 encoded automatically for

whatever value we supply. When u describe the "kubectl describe secret app-secret", values are hidden. However, to retrieve, use
"**kubectl get secret app-secret -o yaml**"

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: bXlzcWw=
  DB_User: cm9vdA==
  DB_Password: cGFzd3Jk
```

echo "Bx6DD11" | base64 --decode (To decode the values)
--> Ways to inject secrets to pod:

- when created as volume, each secret is created as indv., file and content of it is the actual secret value.

ENCRYPTING SECRETS AT REST (encrypting secrets storing in etcd DB)

- Though we encode the secret data, its get stored as plain text in etcd DB.
- apt-get install etcd-client (to install etcdctl)

```
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 \

get /registry/secrets/default/secret1 | hexdump -C
```

- Above gets how the secret is stored in etcd DB. If unencrypted, plain value is shown.
- First check if the encryption is set or not by checking the kube-apiserver service configs.

ps -aux | grep kube-apiserver

- The `kube-apiserver` process accepts an argument `--encryption-provider-config` that specifies a path to a configuration file. The contents of that file, if you specify one, control how Kubernetes API data is encrypted in etcd. If you are running the kube-apiserver without the `--encryption-provider-config` command line argument, you do not have encryption at rest enabled.
- If you are running the kube-apiserver with the `--encryption-provider-config` command line argument, and the file that it references specifies the `identity` provider as the first encryption provider in the list, then you do not have at-rest encryption enabled (**the default `identity` provider does not provide any confidentiality protection.**)
- If you are running the kube-apiserver with the `--encryption-provider-config` command line argument, and the file that it references specifies a provider other than `identity` as the first encryption provider in the list, then you already have at-rest encryption enabled.
- We need to create a EncryptionConfiguration object with resources defined for encryption (like secrets), refer to https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/ and then in the kube-apiserver config setting use `--encryption-provider-config =` /etc/kubernetes/enc/enc.yaml
- In that object file, if identity is defined first; no encryption is done. There are other algorithms available. If others are specified, encryption takes place.
- head -c 32 /dev/urandom | base64 --> use this command and generate a key to use for the chosen any algorithm.

sample file:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
   - resources:
     - secrets
     providers:
     - aescbc:
        keys:
          - name: key
            secret: <BASE 64 ENCODED SECRET>
     - identity: {} # this fallback allows reading unencrypted secrets;
```

- You will need to mount the new encryption config file to the `kube-apiserver` static pod. Here is an example on how to do that:
- Save the new encryption config file to `/etc/kubernetes/enc/enc.yaml` on the control-plane node.
- Edit the manifest for the `kube-apiserver` static pod (this is for the kubeadm setup): `/etc/kubernetes/manifests/kube-apiserver.yaml` so that it is similar to:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 10.20.30.40:443
  creationTimestamp: null
  labels:
    app.kubernetes.io/component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    ...
    - --encryption-provider-config=/etc/kubernetes/enc/enc.yaml  # add this line
    volumeMounts:
    ...
```

```
  - name: enc                       # add this line
    mountPath: /etc/kubernetes/enc      # add this line
    readOnly: true                  # add this line
  ...
volumes:
...
- name: enc                         # add this line
  hostPath:                        # add this line
    path: /etc/kubernetes/enc          # add this line
    type: DirectoryOrCreate
```

- Above is to mount the local volume to the pod. We placed the enc.yaml in /etc/kubernetes/enc folder so mounting the path.
- Now if u see the newly created secrets in etcd db, it gets encrypted. To update for old ones:

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

INIT CONTAINERS:

A Pod can have multiple containers running apps within it, but it can also have one or more init containers, which are run before the app containers are started.
Init containers are exactly like regular containers, except:

- Init containers always run to completion.
- Each init container must complete successfully before the next one starts.
- If a Pod's init container fails, the kubelet repeatedly restarts that init container until it succeeds. However, if the Pod has a `restartPolicy` of Never, and an init container fails during startup of that Pod, Kubernetes treats the overall Pod as failed.
- To specify an init container for a Pod, add the `initContainers` field into the Pod specification, as an array of `container` items (similar to the app `containers` field and its contents).
- If you specify multiple init containers for a Pod, kubelet runs each init container sequentially. Each init container must succeed before the next can run. When all of the init containers have run to completion, kubelet initializes the application containers for the Pod and runs them as usual.

InitContainers

In a multi-container pod, each container is expected to run a process that stays alive as long as the POD's lifecycle. For example in the multi-container pod that we talked about earlier that has a web application and logging agent, both the containers are expected to stay alive at all times. The process running in the log agent container is expected to stay alive as long as the web application is running. If any of them fails, the POD restarts.
But at times you may want to run a process that runs to completion in a container. For example a process that pulls a code or binary from a repository that will be used by the main web application. That is a task that will be run only one time when the pod is first created. Or a process that waits for an external service or database to be up before the actual application starts. That's where **initContainers** comes in.
An **initContainer** is configured in a pod like all other containers, except that it is specified inside a `initContainers` section, like this:

```
apiVersion: v1
kind: Pod
metadata:
name: myapp-pod
labels:
app: myapp
spec:
containers:
- name: myapp-container
image: busybox:1.28
command: ['sh', '-c', 'echo The app is running! && sleep 3600']
initContainers:
- name: init-myservice
image: busybox
command: ['sh', '-c', 'git clone <some-repository-that-will-be-used-by-
application> ; done;']
```

When a POD is first created the initContainer is run, and the process in the initContainer must run to a completion before the real container hosting the application starts.
You can configure multiple such initContainers as well, like how we did for multi-containers pod. In that case each init container is run **one at a time in sequential order**.
If any of the initContainers fail to complete, Kubernetes restarts the Pod repeatedly until the Init Container succeeds.

```
apiVersion: v1
kind: Pod
metadata:
name: myapp-pod
labels:
app: myapp
spec:
containers:
- name: myapp-container
image: busybox:1.28
command: ['sh', '-c', 'echo The app is running! && sleep 3600']
initContainers:
- name: init-myservice
image: busybox:1.28
command: ['sh', '-c', 'until nslookup myservice; do echo waiting for
myservice; sleep 2; done;']
- name: init-mydb
image: busybox:1.28
command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2;
done;']
```

CLUSTER MAINTENANCE :

- If a worker node is down, k8s waits for certain time set as "pod-eviction-timeout" in kube-controller-manager. And if the node is not back till the time, it is considered as dead and pods and recreated in healthy nodes by the replication controller if they're part of replica set.
- kubectl drain node-1 (to purposefully bring down node so that the pods part of replicaset will be recreated on other healthy nodes and updates can be done on node-1; node-1 will be marked as unschedulable ~ cordon)
- kubectl cordon node-2 (to mark a node as unschedulable; no new pods created but holds the existing ones as is)
- kubectl uncordon node-1 (once patched, we have to manually mark it as schedulable to have new pods created on node-1)

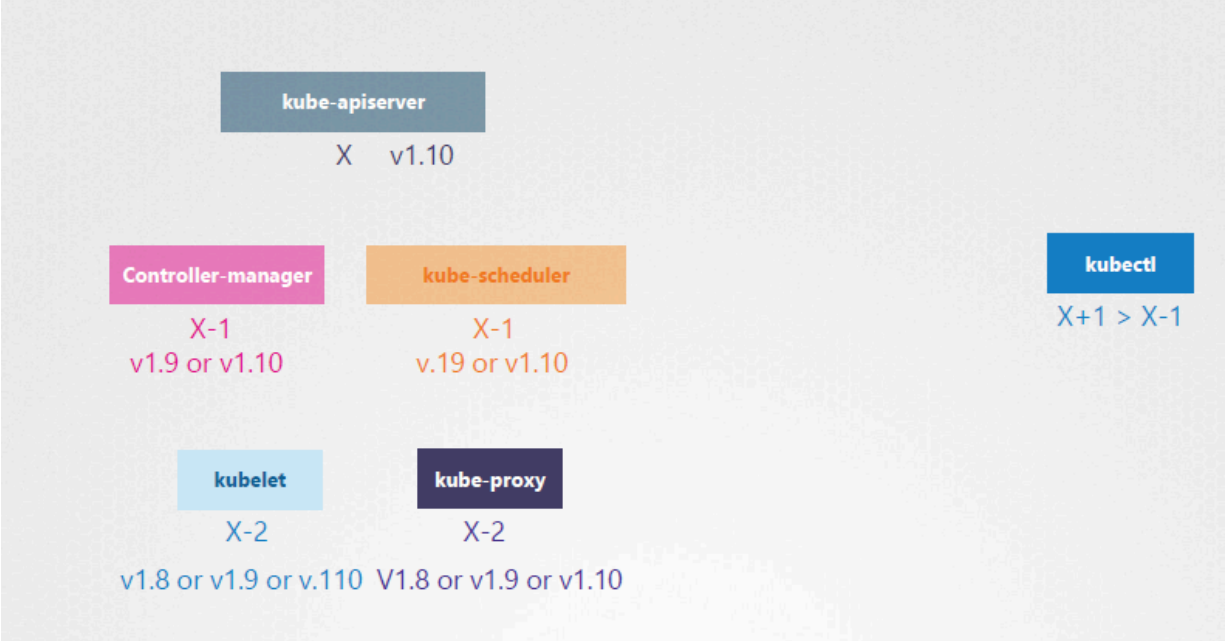
NOTE: If we drain a node and if some pods aren't part of replicaset/ deployment, we will have to force drain the node to bring down all pods. In this case, those pods will be lost forever and

are not created in other healthy nodes.

k8s software version/ cluster upgrade process:

v1, v2... are called major versions
v1.2, v1.0, v2.15 are called minor versions
v1.2.5 (the 5 is called patch version)
however there will be alpha (where the features are disabled) and beta versions (enabled features).

- ETCD cluster and core DNS components are external projects and have their own release cycles. For each release of core k8s components, there will be set of supported versions of these external components.



The permissible range of versions. for kubectl it can be v1.11, v1.10, v1.9

- k8s supports only recent 3 minor releases so before its late, we'll have to upgrade the components. Recommended way is to upgrade to one minor version at a time.

upgrade strategies :
For master --> First bring down master and upgrade it. during this time the existing pods will continue to work, controllers will be down so if a pod is down; new one will not be created, new appln can't be deployed, kubectl doesn't work as api server is down.

1. Next for worker nodes, bring down all nodes and upgrade them.
2. Bring down one by one nodes by marking it as drain and upgrade.
3. Add new node with required version,schedule pods onto it and repeat. (prefered in cloud)

kubeadm method:

- kubeadm upgrade plan (gives the current version and available version to upgrade)
- kubelet has to be upgraded manually for each node. (kubeadm doesn't install / upgrade kubelet) and kubeadm tool also has to be upgraded which follows same release cycle as that to k8s

STEPS:
--> cat /etc/*release* - gives the distribution type details of the nodes.
FIRST MASTER NODE -->

1. apt-get upgrade -y kubeadm=1.12.0-00
2. kubeadm upgrade apply v1.12.0 (this upgrades the kubernetes components)

If u run kubectl get nodes now, it will show the old version as this command gets the output of kubelets. (at this point kubelet is still not upgraded)
If master node has kubelet installed, only then master node will be shown in k get nodes command and hence we need to upgrade kubelet on master.

3. apt-get upgrade -y kubelet=1.12.0-00
4. systemctl restart kubelet

NOW WORKER NODES -->

1. kubectl drain node-1
2. apt-get upgrade -y kubeadm=1.12.0-00
3. kubeadm upgrade apply v1.12.0
4. kubeadm upgrade node config --kubelet-version v1.12.0
5. systemctl restart kubelet
6. kubectl uncordon node-1

REPEAT same on other nodes.

demo:
To seamlessly transition from Kubernetes v1.30 to v1.31 and gain access to the packages specific to the desired Kubernetes minor version, follow these essential steps during the upgrade process. This ensures that your environment is appropriately configured and aligned with the features and improvements introduced in Kubernetes v1.31.

On the controlplane node:

Use any text editor you prefer to open the file that defines the Kubernetes apt repository.

```
vim /etc/apt/sources.list.d/kubernetes.list
```

Update the version in the URL to the next available minor release, i.e v1.31.

```
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
https://pkgs.k8s.io/core:/stable:/v1.31/deb/ /
```

After making changes, save the file and exit from your text editor. Proceed with the next instruction.

```
apt update

apt-cache madison kubeadm
```

Based on the version information displayed by apt-cache madison, it indicates that for Kubernetes version 1.31.0, the available package version is 1.31.0-1.1. Therefore, to install kubeadm for Kubernetes v1.31.0, use the following command:

```
apt-get install kubeadm=1.31.0-1.1
```

Upgrade the node --> this is for worker node instead of kubeadm upgrade apply v1.31.0

kubeadm upgrade node

Run the following command to upgrade the Kubernetes cluster.

```
kubeadm upgrade plan v1.31.0

kubeadm upgrade apply v1.31.0
```

Note that the above steps can take a few minutes to complete.
Now, upgrade the Kubelet version. Also, mark the node (in this case, the "controlplane" node) as schedulable.

```
apt-get install kubelet=1.31.0-1.1
```

Run the following commands to refresh the systemd configuration and apply changes to the Kubeletservice:

```
systemctl daemon-reload

systemctl restart kubelet
```

Refer: https://v1-30.docs.kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/

BACKUP AND RESTORE:

- If we strictly use only declarative way of creating components, the better way for backup is to have the yaml files in a centralised SCM.
- If unsure, we can backup using "kubectl get all --all-namespaces -o yaml > all-deploy-services.yaml" (by calling kubeapi server)

backing up ETCD:
ETCD DB stores all the info about cluster.
while u install etcd, there will be a --data-dir config where the data is stored in the master nodes by etcd.
etcdctl comes with option to take snapshots and restore from them.
--> ETCDCTL_API=3 etcdctl snapshot save snapshot.db
--> ETCDCTL_API=3 etcdctl snapshot status snapshot.db
To restore, etcd will be restarted in the process and we'll have to stop kube api server as it depends on etcd.
--> ETCDCTL_API=3 etcdctl snapshot restore snapshot.db --data-dir /var/lib/etcd-from-backup

when we restore, etcd initialises new cluster config, creates new cluster components to a new cluster, so as to prevent new components joining existing cluster. In above case, new data dir is created and we'll have to configure same to etcd service:

--data-dir=/var/lib/etcd-from-backup
--> systemctl daemon reload
--> service etcdctl restart
--> service kube-apiserver start

- export ETCDCTL_API=3; to set the variable for the etcdctl api version

ex commands:
ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \
--cacert=/etc/kubernetes/pki/etcd/ca.crt \
--cert=/etc/kubernetes/pki/etcd/server.crt \
--key=/etc/kubernetes/pki/etcd/server.key \
snapshot save /opt/snapshot-pre-boot.db

--> The restore will not require any cert/ keys bcoz we operating command in our local and not related to connecting to etcd db. No issue though we specify.

- If the etcd is running as pod (normal or static), its called as stacked etcd. If on external server, called as external etcd.
- For a external running etcd, to restore we will need to have the backup file on it and then use restore command to create new data dir. Update the same to the etcd server (usually /etc/system.d/system/etcd.service) then:
  --> systemctl daemon reload
  --> systemctl restart etcd
- Its recommended to restart controller, scheduler pods to not have them running on stale data; restart kubelet process.

NETWORKING

authentication:

--> In a simple way, we can add password,username,userID in a csv file and provide this info to the apiserver pod startup command like:
spec:
containers:
- command:
- kube-apiserver
- --authorization-mode=Node,RBAC
<content-hidden>
- --basic-auth-file=/tmp/users/user-details.csv

so while users call api calls, they can pass [curl -v -k https://localhost:6443/api/v1/pods -u "user1:password123"] to authenticate.

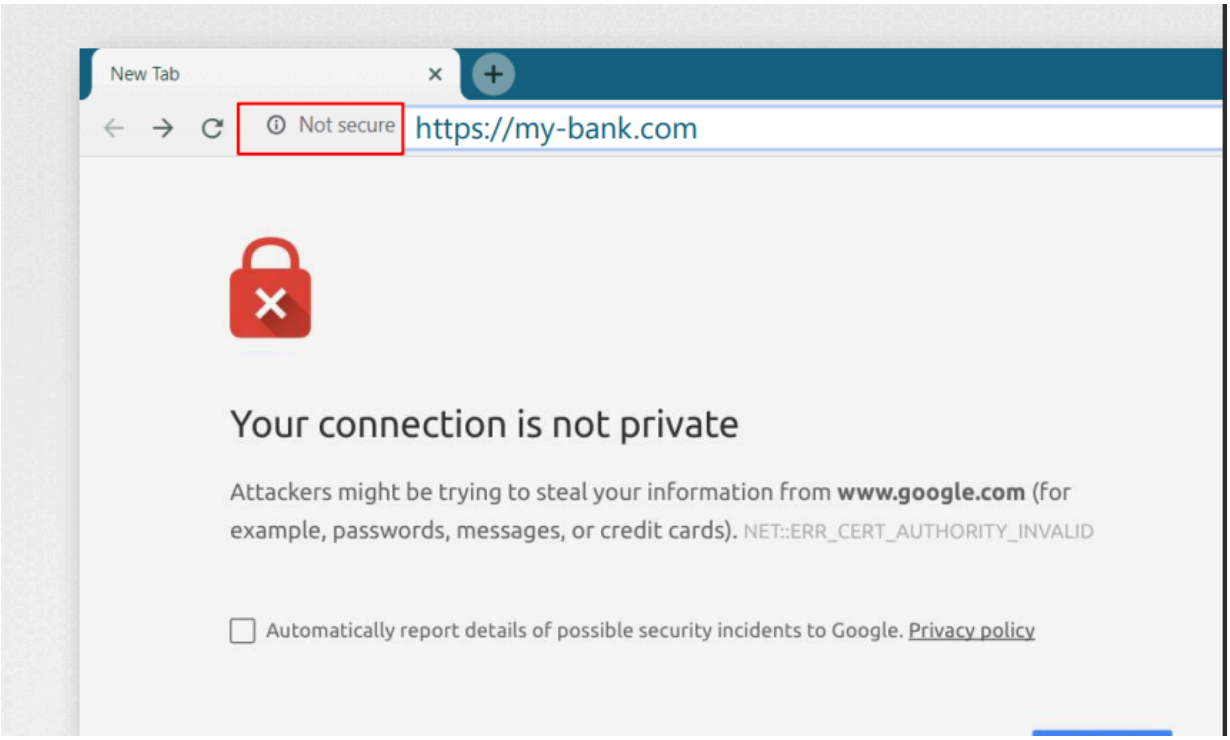
--> This can also be done via tokens, csv file having authtoken,username,userID and pass to apiserver config:
"--token-auth-file=/var/token_user.csv" so while users call api calls, they can pass "authorization : Bearer <token>" to authenticate.
--> U will also need to create roles and role bindings to the users to complete the process.

TLS:
symmetric encryption: using single key to encrypt and decrypt.
--> If users credentials have to be transferred from the client to the server, user will encrypt the data using a key and send the key, data to the server for decryption. But there's a chance of hackers hacking the key and data, so this is unsafe.
Asymmetric encryption: using 2 keys, one to encrypt and other to decrypt.
--> In above case, To safely transfer the symmetric key from client to server, we can use asymmetric encryption to encrypt the symmetric key.
--> First the server generates public and private rsa key and sends the pub key to the client.
"openssl genrsa -out my-bank.key 1024" --> generates both keys
"openssl rsa -in my-bank.key -pubout > my-bank.pem" --> to extract public key from above
--> Now when the user's browser access https site of the bank, it sends its pub key. The client browser encrypts the symm key with this public key and transfers same to the bank server. As the bank server has the private key with it, it can decrypt the data and fetch the symm key which will be used for further safe communication. THE SYMM KEY IS SAFELY TRANSFERRED.
--> From now, the user can encrypt his credentials with the symm key and transfer to the server. Therefore only client and server has the symm key. THIS IS SAFE TRANSACTION.
Note: Though the hacker sniffs the encrypted symm key and the data (encrypted with symm key), he cannot decrypt as the private key used to encrypt symm key is only with the bank.

- For ssh connection, if we want to restrict, we will ask users to generate ssh-keygen keys and add their public keys on the server "~/.ssh/authorized_keys" file. And the user can login using his private key " ssh -i id_rsa ubuntu@10.20.4.7"

what if hacker finds other ways to hack ur credentials ? what if he creates exact same copy of the bank server in https:// format and mimics above safe transaction case?
--> This is not actually possible because --> whenever we use a https website, our browser verifies the certificate of the website and checks its certificate authority's public key to ensure

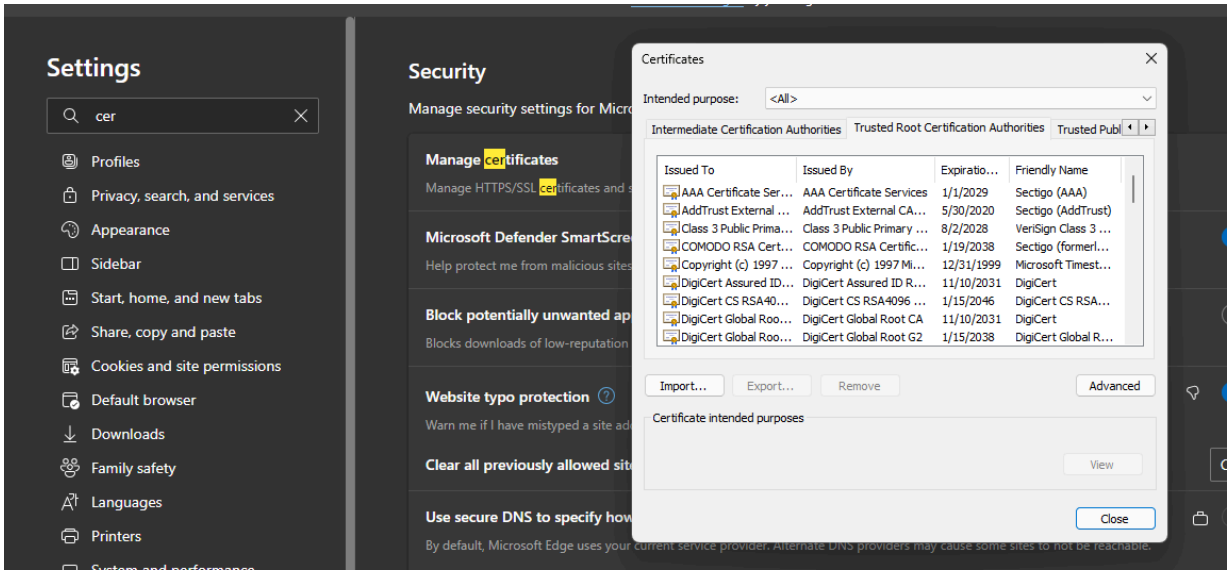
if its a right CA and the cert is legit.
--> Our browser has the mechanism to validate the CA's public key and check if the site is secure or not and warns accordingly.



--> we can send a CSR to the CA's to create a legitimate cert for a website. "openssl req -new -key my-bank.key -out my-bank.csr -subj "/C=US/ST=CA/O=MyOrg, Inc./CN=my-bank.com". Now once the CA validates the CSR, issues the cert.
--> If the hacker sends the cert req to the CA claiming as the my-bank.com, he fails at the validation phase.

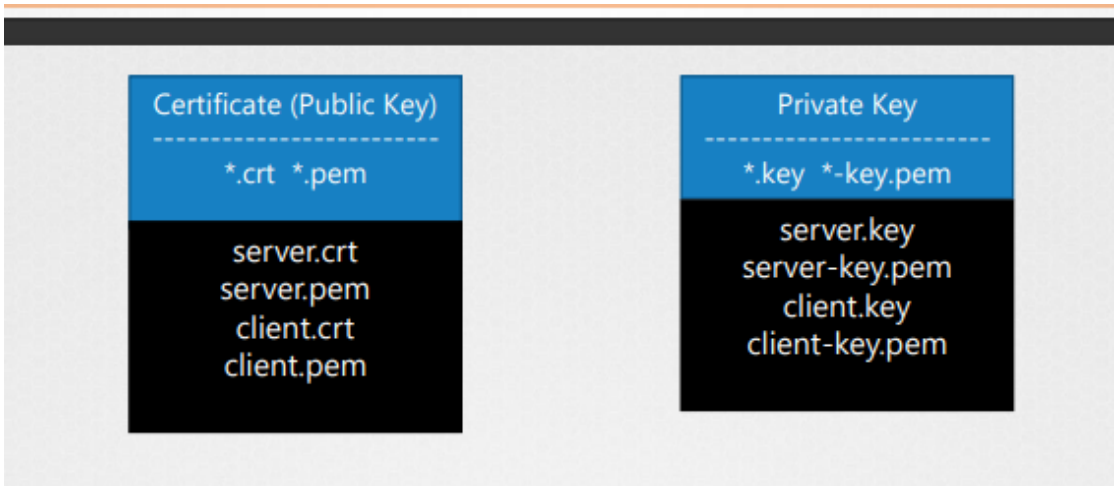
- How does browsers know if the CA is the right CA and not someone who claims they are ?

The CA's sign the cert using their pvt key and our browsers have the public keys of the CA's. and hence they can validate the CA's using pub key.



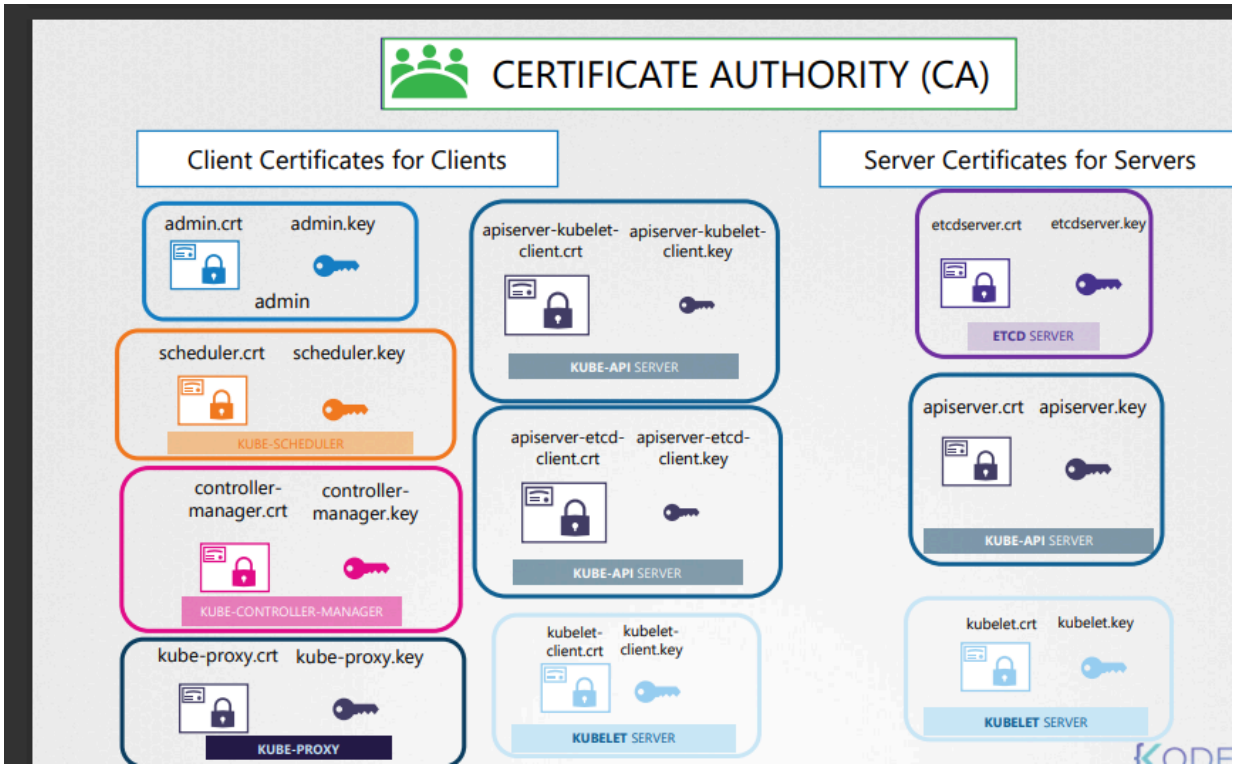
--> These CA's also have private offerings useful to host secure websites used within our org. Like in IT company's payroll portals. We can add that particular public key in all of the employees browsers to have the secure and trusted connection to the website.

naming conventions :



TLS IN KUBERNETES:

- All the other k8s components and users talk to kubeapi server so they should present their client certs.
- kubeapi talks to etcd; kubeapi talks to kubelet so in this case kubeapi is client and etcd, kubelet are servers so accordingly they should present their certs.



cert creation process:

- openssl genrsa -out ca.key 2048
- openssl req -new -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr
- openssl x509 -req -in ca.csr -signkey ca.key -out ca.crt (here we are self signing the CA cert using the same key we generated for the csr itself)

--> we can now use this ca.crt to sign other certs like:

- openssl genrsa -out admin.key 2048
- openssl req -new -key admin.key -subj "/CN=KUBE-ADMIN/O=system:masters" -out admin.csr (we add /O=system:masters to add this users to the masters group)
- openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -out admin.crt (here we are signing client cert using CA)

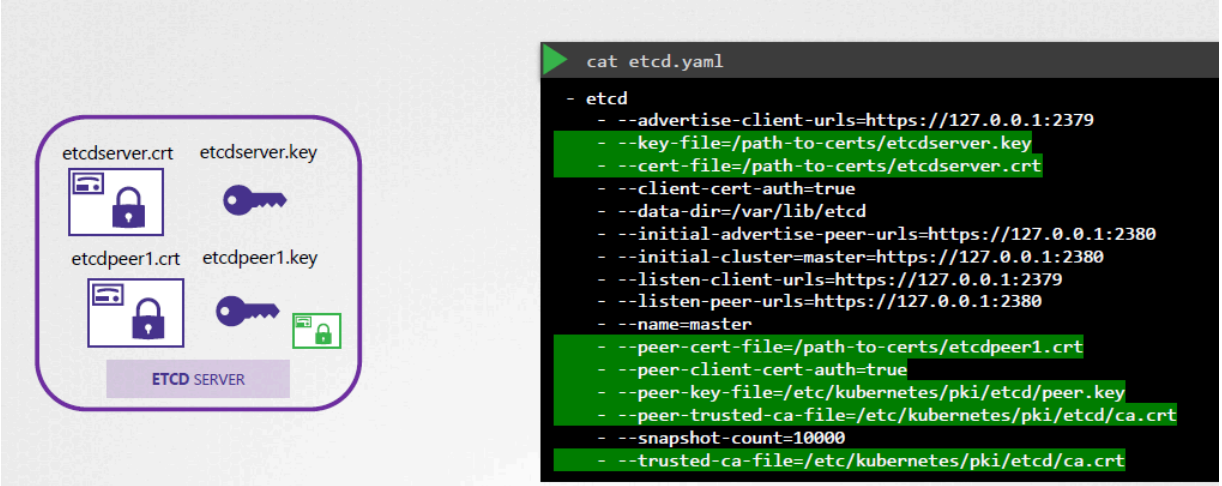
--> for the admins, the system components (scheduler, controller, kube-proxy) we will add system to the cert naming. like system-kube-proxy; system-kube-scheduler; system-kube-controller-manager.

Now when the admin user authenticates, he can use:

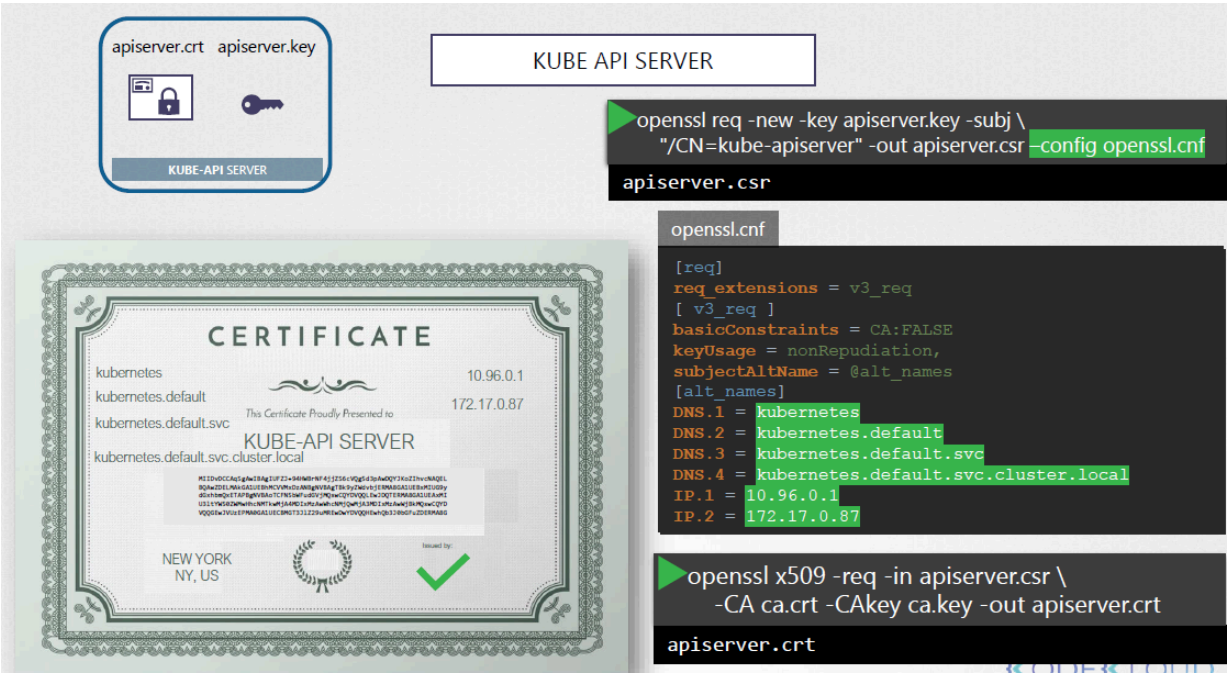
curl https://kube-apiserver:6443/api/v1/pods \ --key admin.key --cert admin.crt --cacert ca.crt

Server certs:

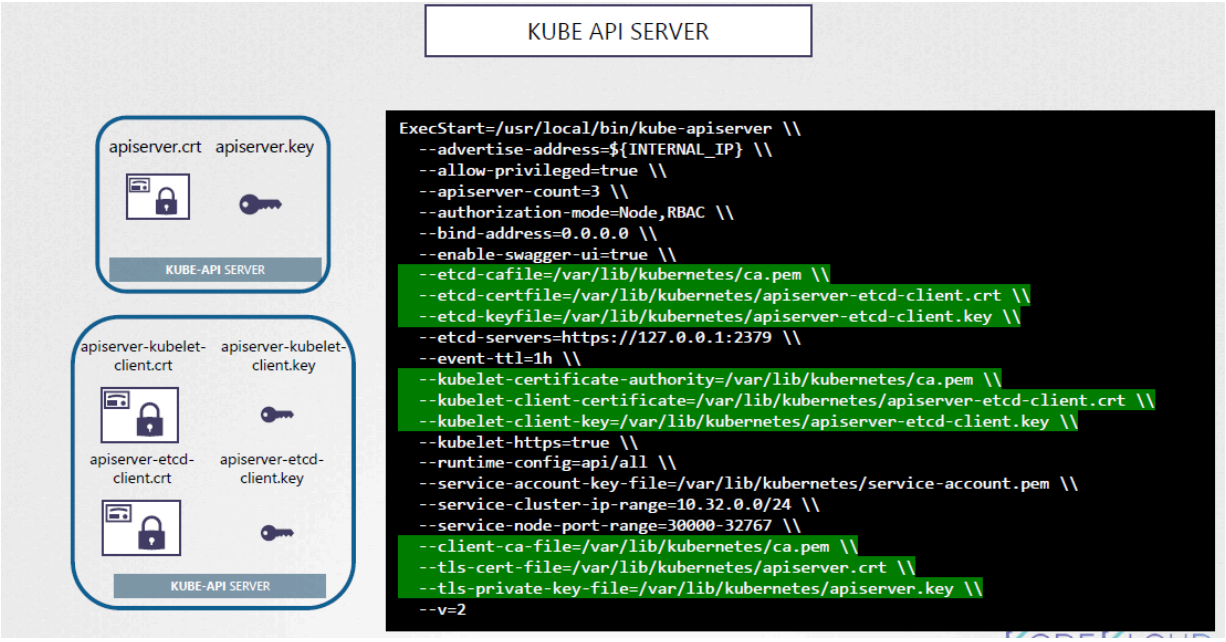
- etcd server will also generate the certs in same way, if there is a etcd cluster method and has other peers, those can also generate certs and add config as below:



--> kube-apiserver is the most used one and usually will be called by alternative names so we should create cnf file and pass the same while creating csr.

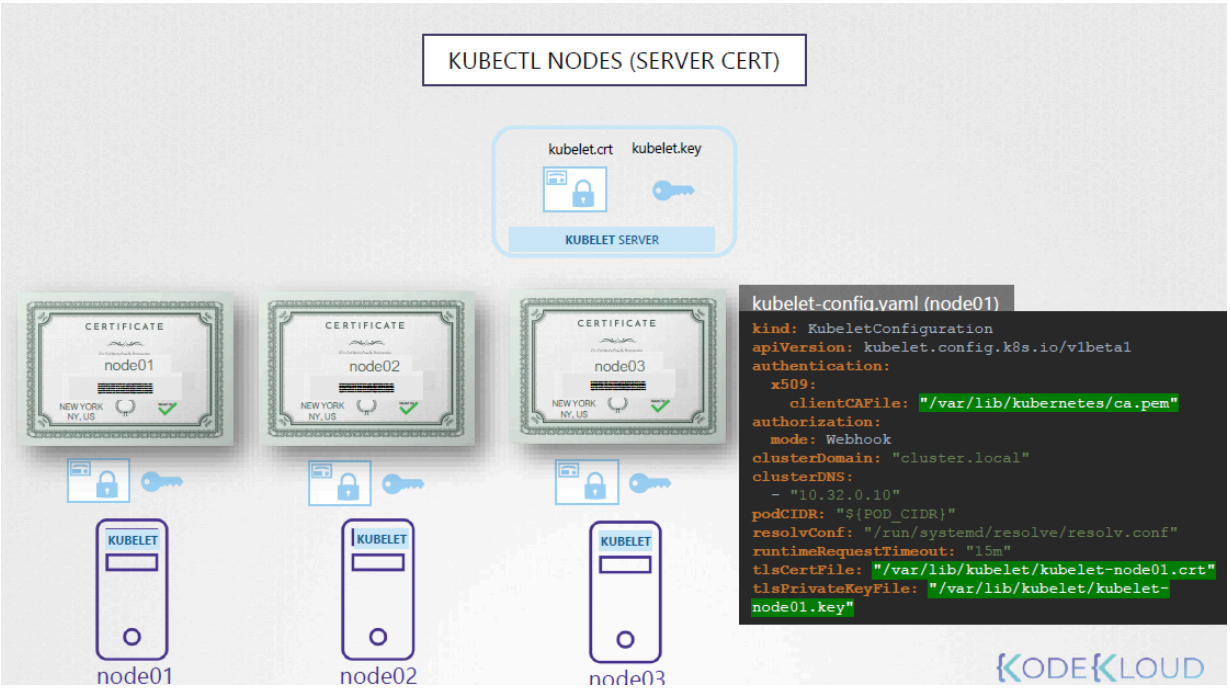


and add the kube-apiserver's server cert, the client certs kube-apiserver uses to connect to the etcd and kubelets.

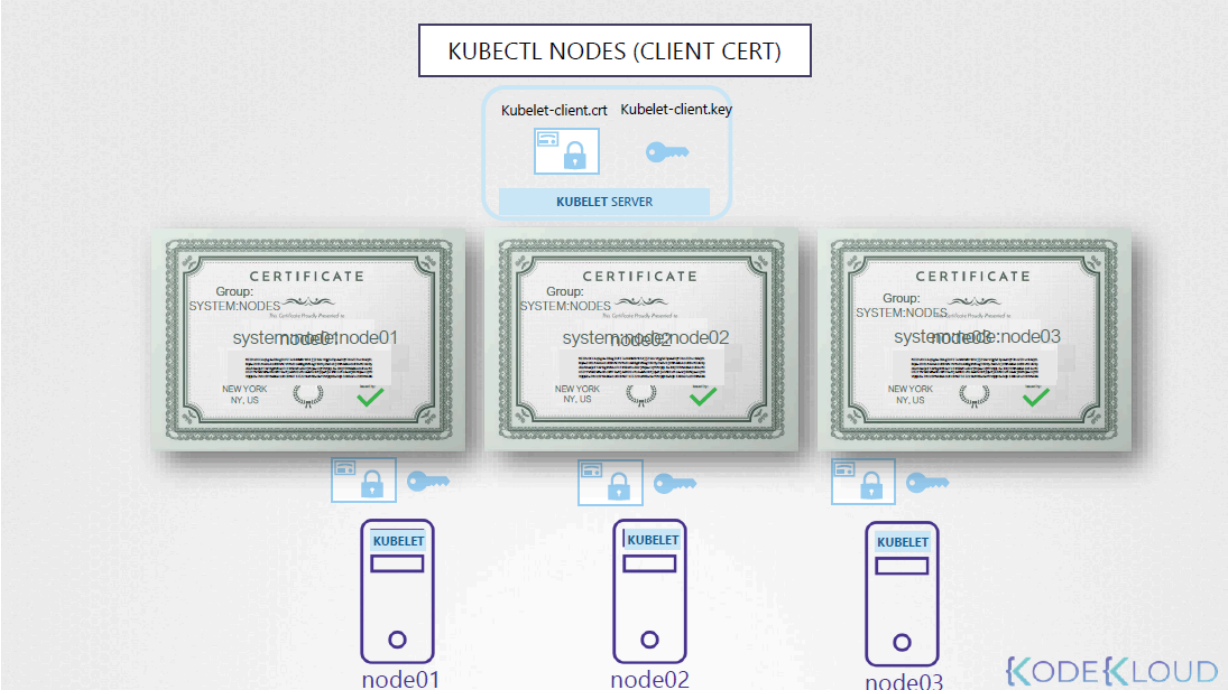


--> kubelet will have to generate server and client certs (server when api-server talks to kubelet and client when kubelet talks to api-server)

--> each node will name its own server cert by the node name - node01/ node02 so that the api-server can identify and add the same details to the kubelet config file. ??



--> As clients, certs will have to name as "system:node:node01" and add to the group "system:nodes" just like we did for the admin users "/O=system:nodes" during csr generation.



To get the details of any cert use " openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout"

--> To troubleshoot issues, u can check the "kubectl logs etcd-master" or check the native running service (if the components are installed hard way- manually)

--> If incase core components needed for the kubectl to work (kube-apiserver and etcd are down), use docker ps and docker logs to get logs from the container.

certificate API in k8s:

--> CA server is basically pair of keys and cert files. in our case, CA server is master.

--> k8s has built in certificate api hosted on **kube-controller manager.** we can use kubectl commands to approve csr.

- openssl genrsa -out jane.key 2048 (cert for a new admin - jane)
- openssl req -new -key jane.key -subj "/CN=jane" -out jane.csr
- Now create a CertificateSigniningRequest object manifest file where the csr file is base64 encoded "cat jane.csr | base64 -w 0" to have the encoded csr in one line
  add the encoded cert under spec: request section.

---->

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: jane
spec:
  expirationSeconds: 600 #seconds
```

```
groups:
 - system:authenticated
usages:
 - digital signature
 - key encipherment
 - server auth
request:
```

--> kubectl get csr (to view all csrs)
--> kubectl certificate approve jane (once approved we can view cert from the csr object again)
--> kubectl get csr jane -o yaml, decode the text content and share to the user.
The kube-controller manager tc of the cert approving but for that it needs the CA root cert and the private key, those configs are stored in the controller-manager yaml file:

```
cat /etc/kubernetes/manifests/kube-controller-manager.yaml
spec:
  containers:
  - command:
    - kube-controller-manager
    - --address=127.0.0.1
    - --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt
    - --cluster-signing-key-file=/etc/kubernetes/pki/ca.key
    - --controllers=*,bootstrapsigner,tokencleaner
    - --kubeconfig=/etc/kubernetes/controller-manager.conf
    - --leader-elect=true
    - --root-ca-file=/etc/kubernetes/pki/ca.crt
    - --service-account-private-key-file=/etc/kubernetes/pki/sa.key
    - --use-service-account-credentials=true
```

- we can deny the csr "kubectl certificate deny <csr-name>"

USAGE OF CLIENT CERTS TO ACCESS K8S:

- Once we have the required client cert, client key and ca cert, we can use these for authentication when we call kube-apiserver or use kubectl commands.

```
curl https://my-kube-playground:6443/api/v1/pods \
  --key admin.key
  --cert admin.crt
  --cacert ca.crt
```

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/pods",
  },
  "items": []
}
```

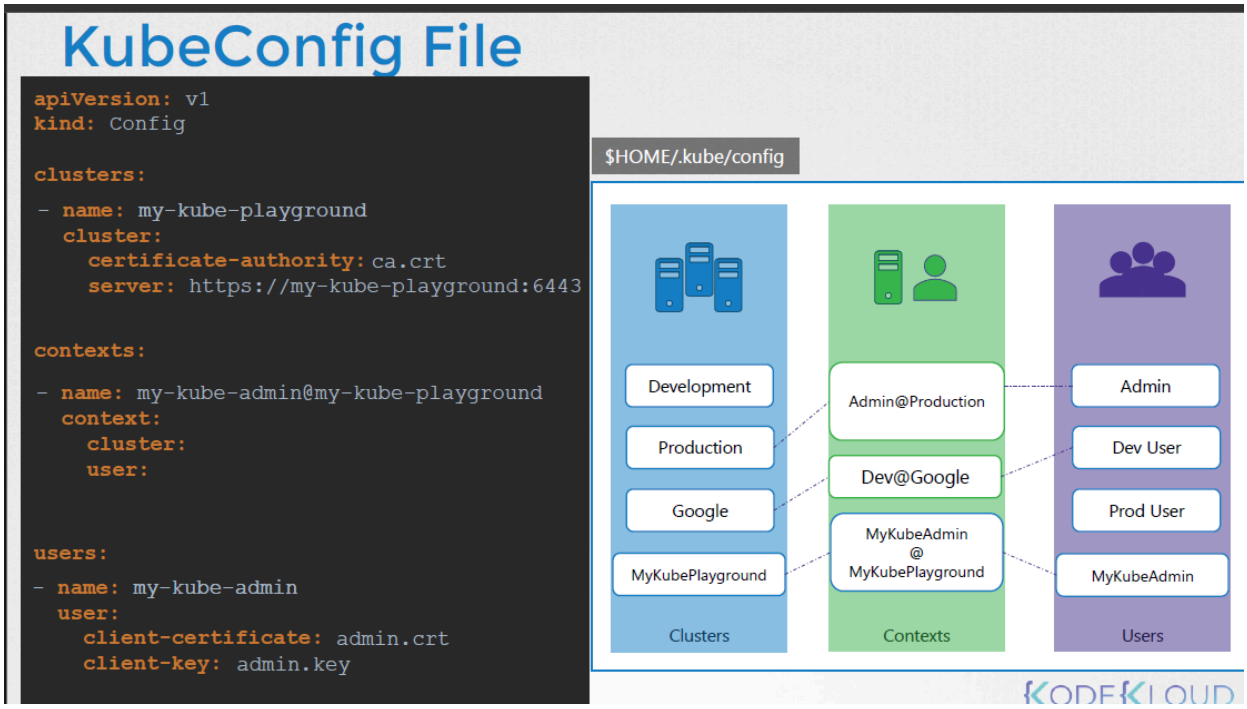
```
kubectl get pods
     --server my-kube-playground:6443
     --client-key admin.key
     --client-certificate admin.crt
     --certificate-authority ca.crt
```

```
No resources found.
```

- But we generally don't specify explicitly, kube config file does that for us.

KUBE CONFIG FILE:

- It consists of 3 sections - clusters, users and contexts.



context is like a matching of which user to use to connect to which cluster.

- "kubectl config view" to get the default kube config file details.
- kubectl config view --kubeconfig=my-custom-config
- kubectl config --kubeconfig=/root/my-kube-config use-context research

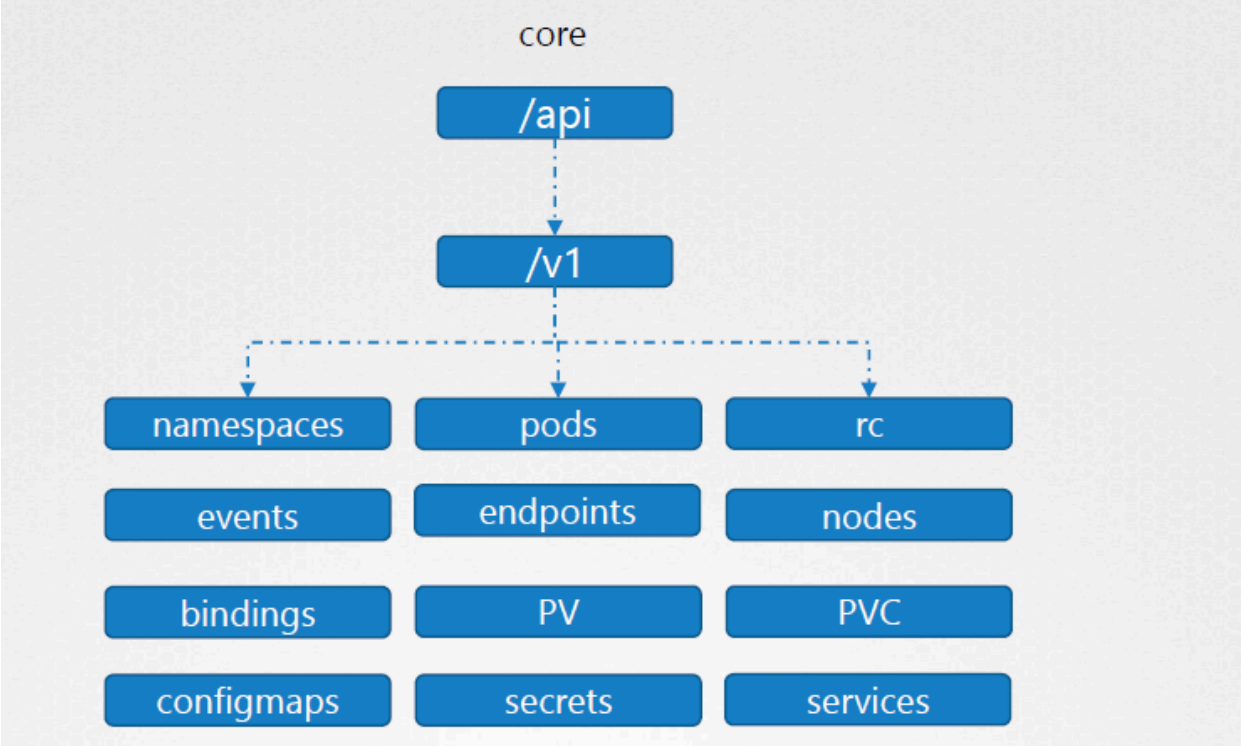
--> To update the default kubeconfig file, vi ~/.bashrc and add the line "export KUBECONFIG=/root/my-kube-config" and then source the bashrc to have changes applied to current shell session : source ~/.bashrc

- we can add namespace field under context spec for a particular context to use that ns by default.
- Recommended way to specify the certs is to add the base64 encoded certs to the kube config file. As:

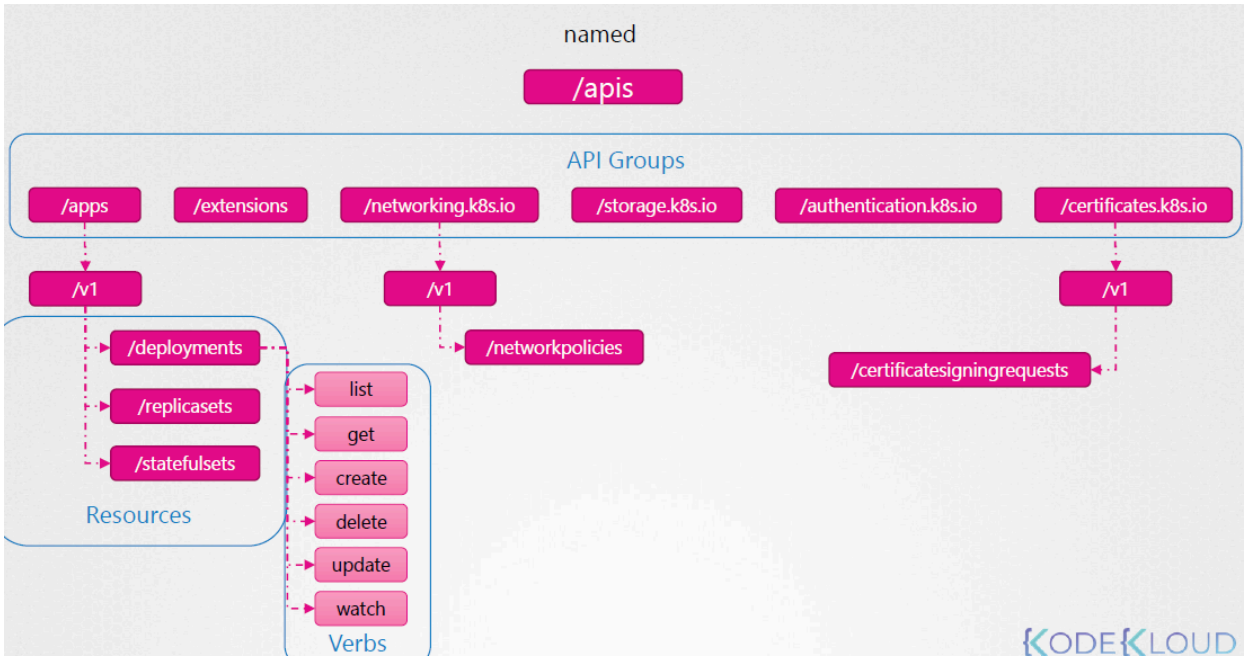
"certificate-authority-data: <base-64 encoded cert>"

API GROUPS:

All resources in k8s are grouped into diff api groups. At the top level we have /api (core) and /apis (named).



- Under named api group, we have diff groups. and each has their own set of resources as shown below:
- And the actions we can do on the resources are called as verbs.



- To list the api groups, use "curl http://localhost:6443 -k" and to list resources under an api group, use [curl http://localhost:6443/apis -k | grep “name”]
- But for above to work, we need to specify ca, client cert and client key to authenticate ourself. If we want to use the kubeconfig to authenticate us, use "kubectl proxy" (NOTE KUBECTL PROXY != KUBE PROXY, here it is kubectl proxy)command so a new kubectl proxy client will run on 8001 port and now u can directly use "curl http://localhost:8001 -k" without specifying certs.

AUTHORIZATION:

- Once we are authenticated, what we are permitted to do is called as authorization. There are diff methods for that:

1. Node authorizer
2. RBAC
3. ABAC (attribute)
4. webhook
5. always allow (without checking it allows for all, this is the default one)
6. always deny

- Node authorizer: This is specially for the kubelet to authorize. The Node authorizer allows a kubelet to perform API operations. In order to be authorized by the Node authorizer, kubelets must use a credential that identifies them as being in the `system:nodes` group, with a username of `system:node:<nodeName>`

- ABAC: This is creating access policy in json format for a user/ set of users and passing this config for apiserver. each time there is a change to this file, apiserver has to be restarted. (This is not a topic to stress)
  {"kind": "Policy", "spec": {"user": “dev-user", "namespace": "*", "resource": “pods", "apiGroup": "*"}}
- RBAC: we create roles with set of permissions and associate users/ groups to this roles.
- Webhook: This is by using 3rd party tool. whenever any user requests, kubernetes makes api call to this 3rd party and based on its response, the user will be granted access.

we will define the mode(s) in kube apiserver config as "--authorization-mode=Node,RBAC,Webhook". If multiple modes defined, request gets checked one by one in the same order and once a module rejects passed to other until one module approves.

RBAC IN DETAIL:

creating a role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: Developer
rules:
 - apiGroups: [""] # "" indicates the core API group
   resources: ["pods"] # this resource name is from the perspective of http calls
   verbs: ["list", "get", "create", "update", "delete" ]
   resourceNames: ["blue", "green"]
 - apiGroups: [""]
   resources: ["ConfigMap"]
   verbs: ["create"]
```

creating a role binding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: Developer-RoleBinding
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
```

```
# "roleRef" specifies the binding to a Role / ClusterRole
kind: Role #this must be Role or ClusterRole
name: Developer # this must match the name of the Role or ClusterRole you wish to bind to
apiGroup: rbac.authorization.k8s.io
```

- kubectl get roles
- kubectl get rolebindings
- kubectl auth can-i create deployments
- kubectl auth can-i delete pod
- kubectl auth can-i delete pod --as Developer / kubectl delete pod --as Developer
- kubectl auth can-i create deployment --as Developer --namespace blue
- To create a Role:- `kubectl create role developer --namespace=default --verb=list,create,delete --resource=pods`
- To create a RoleBinding:- `kubectl create rolebinding dev-user-binding --namespace=default --role=developer --user=dev-user`
- kubectl api-resources (gets details of all the api resources)
- kubectl api-resources --namespaced=true (to get the api components which are namespace scoped) like pods, replica sets, services.
  cluster scoped are - nodes, pv, pvc, cert signing requests, namespaces.
- we need to create cluster role and cluster role bindings for the resources which are cluster scoped as mentioned above.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
name: cluster-administrator
rules:
- apiGroups: [""]
resources: [“nodes"]
verbs: ["list“, "get", “create“, “delete"]

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
name: cluster-admin-role-binding
subjects:
- kind: User
name: cluster-admin
```

apiGroup: rbac.authorization.k8s.io
roleRef:
kind: ClusterRole
name: cluster-administrator
apiGroup: rbac.authorization.k8s.io

NOTE: if we create "cluster role" type of role for pods kind and bind it to a user, the user will have access defined across the whole cluster.

SERVICE ACCOUNTS: We use SA for any external applications to authorize to kubernetes api ex., prometheus, jenkins.

- kubectl create serviceaccount dashboard-sa
- kubectl create token <sa-name>

- By default, for each namespace, a default service account is created which has minimal permissions and If you deploy a Pod in a namespace, and you don't manually assign a ServiceAccount to the Pod, Kubernetes assigns the `default` ServiceAccount for that namespace to the Pod.
- To create a token for the service account, "kubectl create token <sa-name> : by default this will expire in 1hr, we can specify to increase time.
- To have custom token mounted to pod, use : serviceAccount: <sa-name> in the pod definition file.
- just like for user role binding and cluster role binding, we will create one role and create binding for it in the type:ServiceAccount.

IMAGE SECURITY:
when image: nginx is specified, it actually is : docker.io/library/nginx where docker.io is the dns name for the dockerhub registry, library is the default account.

- docker login <registry-name> (to login to a registry command line wise)
- But in runtime, we will need to create secret object of type "dockerregistry" and use this in pod spec: imagePullSecrets as below.

kubectl create secret docker-registry regcred --docker-server=<registryname> --docker-username=<> --docker-email=<> --docker-password=<>

- Here the type of secret SHOULD BE docker-registry.

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

```
  name: nginx-pod
spec:
  imagePullSecrets:
  - name: regcred
  containers:
  - name: nginx
    image: privateregistry.io/radha/nginx
```

DOCKER SECURITY:

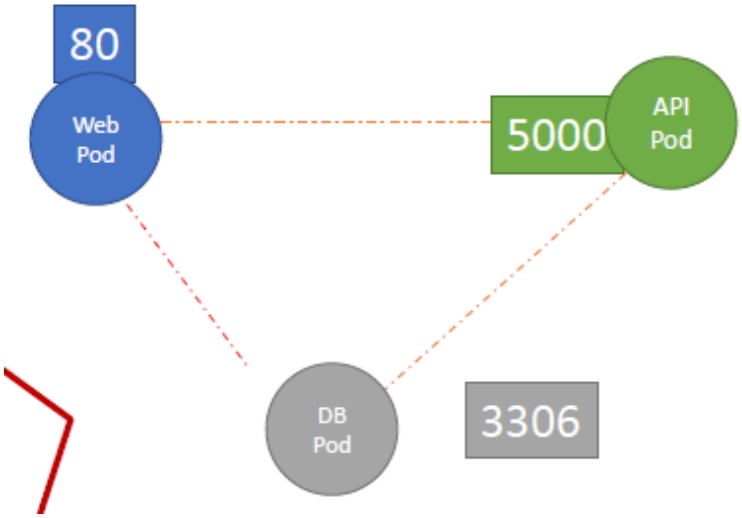
- On a host having docker runtime installed, the processes ran by the docker-containers will be on a separate namespace wrt to the other host processes. This is for the isolation.
- When u list the process from the host, u will be able to view the container process as well. But the opposite is not possible from the container.
- By default, docker runs the container processes as root user (but this user will not have all privileges as that of host root user).
- To change the user with which container process should run: docker run --user=1000 ubuntu sleep 3600.
- To add or remove permission to the root user in container: docker run --cap-add MAC_ADMIN ubuntu / docker run --cap-drop MAC_ADMIN ubuntu / docker run --privileged ubuntu.

- we can have same implementation in k8s as well either in pod level (applies to all containers in it) or container level. NOTE: if pod level and container level security settings are applied, container level one overrides the pod level settings.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
    securityContext:
      runAsUser: 1000
      capabilites:   #here capabilities are applied only at container level.
        add: ["MAC_ADMIN"]
```

NETWORK POLICIES:

If we have 3 pods and want to restrict ingress traffic to the db pod only from API pod on 5000 port and not from anyothers, we can create a network policy.

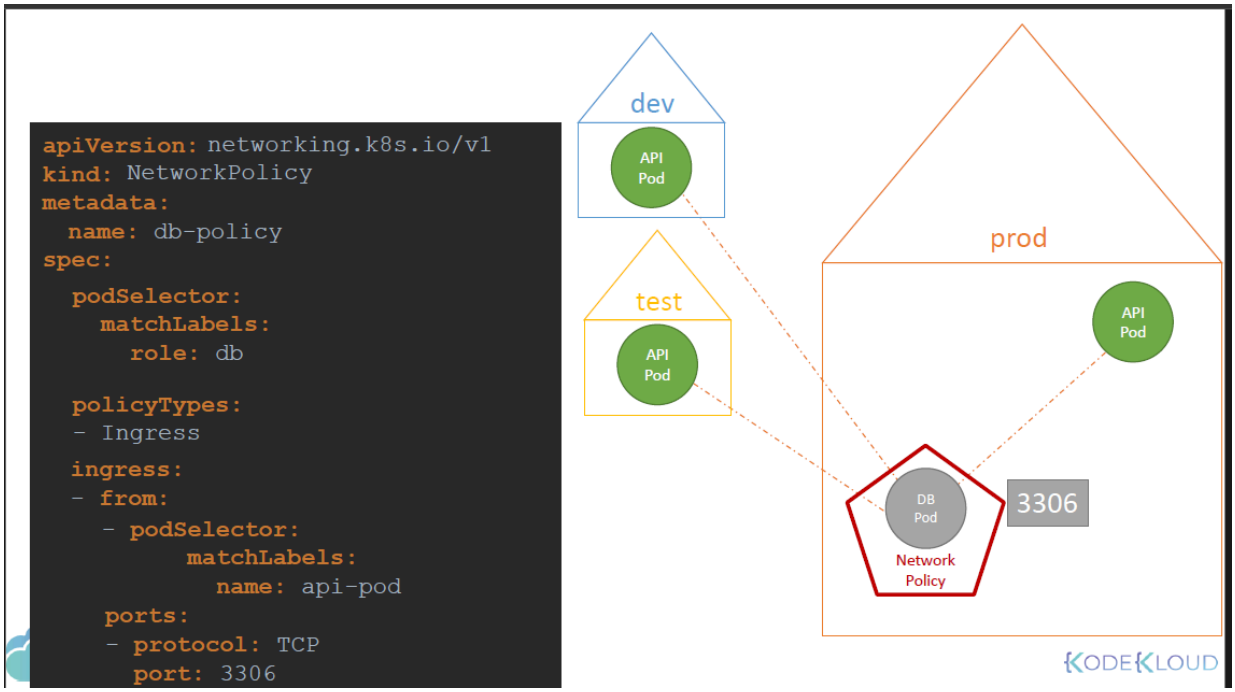


sample file to allow ingress to db pod only from api pod: Here no Egress traffic is allowed (Egress traffic is the one originating from the db-pod; Egress doesn't mean the response we sent from db to api pod in the response.)

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          name: api-pod
    ports:
    - protocol: TCP
      port: 3306
```

NOTE: There will be some network solutions which doesn't allow network policy objects. i.e., though the NP object is created, the changes will not be effective.
Also when we allow ingress traffic to db pod <-- from API pod, the response sent from db pod --> to api pod is automatically allowed. So we just need to worry about ingress traffic ; the actual traffic.

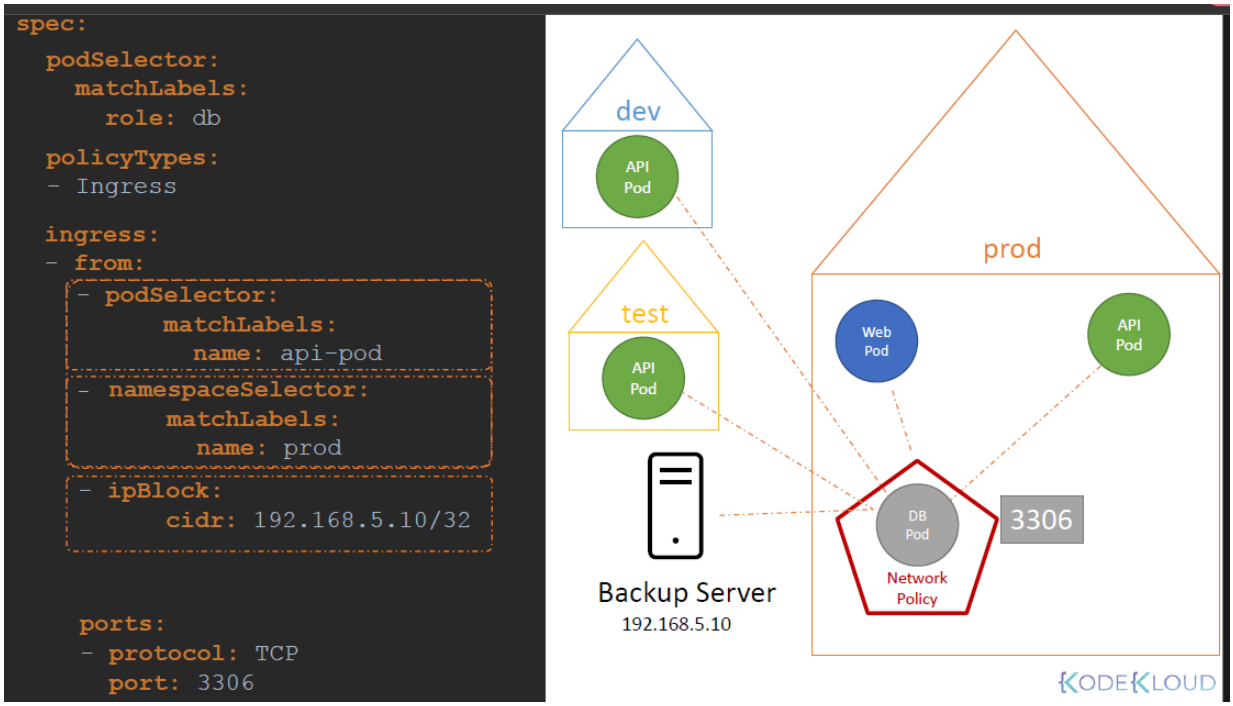
EX: 1 to allow traffic from pods having label: name:api-pod from any namespaces.



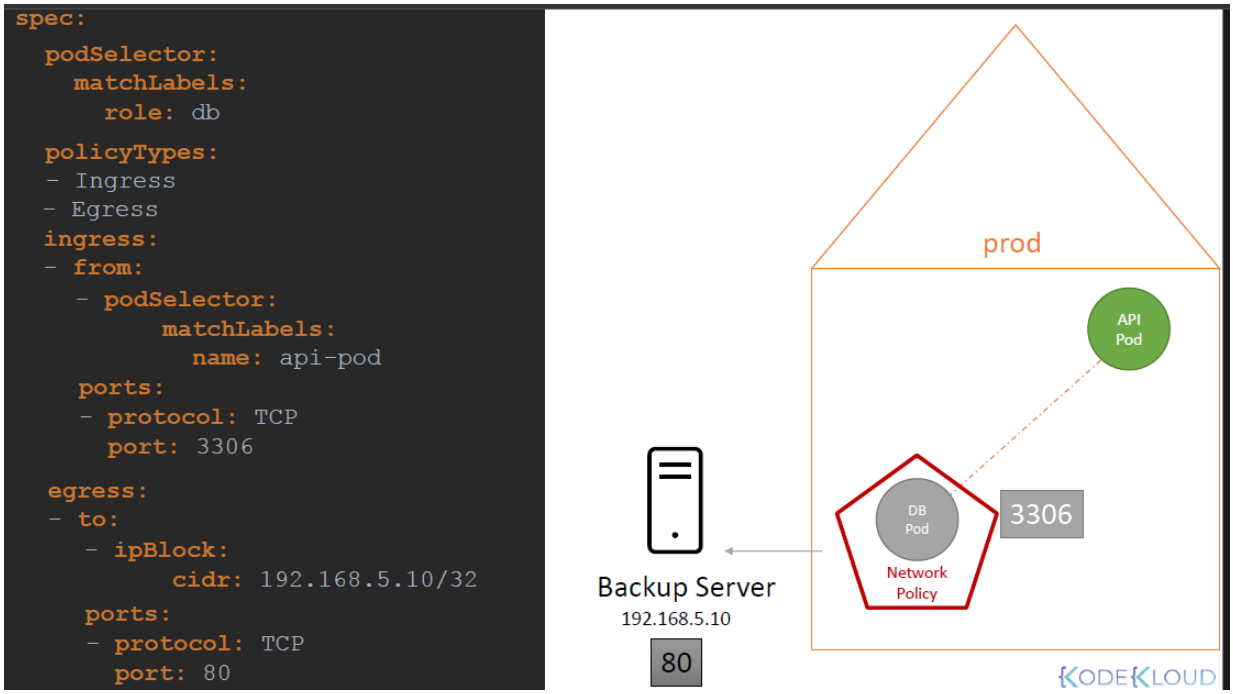
EX: 2, allows traffic to db pod only from api-pod in the prod namespace. (this is like AND operation: the pod label should be api-pod AND it should be from prod namespace)

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          name: api-pod
      namespaceSelector:
        matchLabels:
          name: prod
    ports:
    - protocol: TCP
      port: 3306
```

EX: 3 all 3 are specified as OR operator type...so traffic allowed either from all pods in prod ns OR pods with the label OR from the external server with the IP range. (exception: traffic to and from the node where a Pod is running is always allowed)



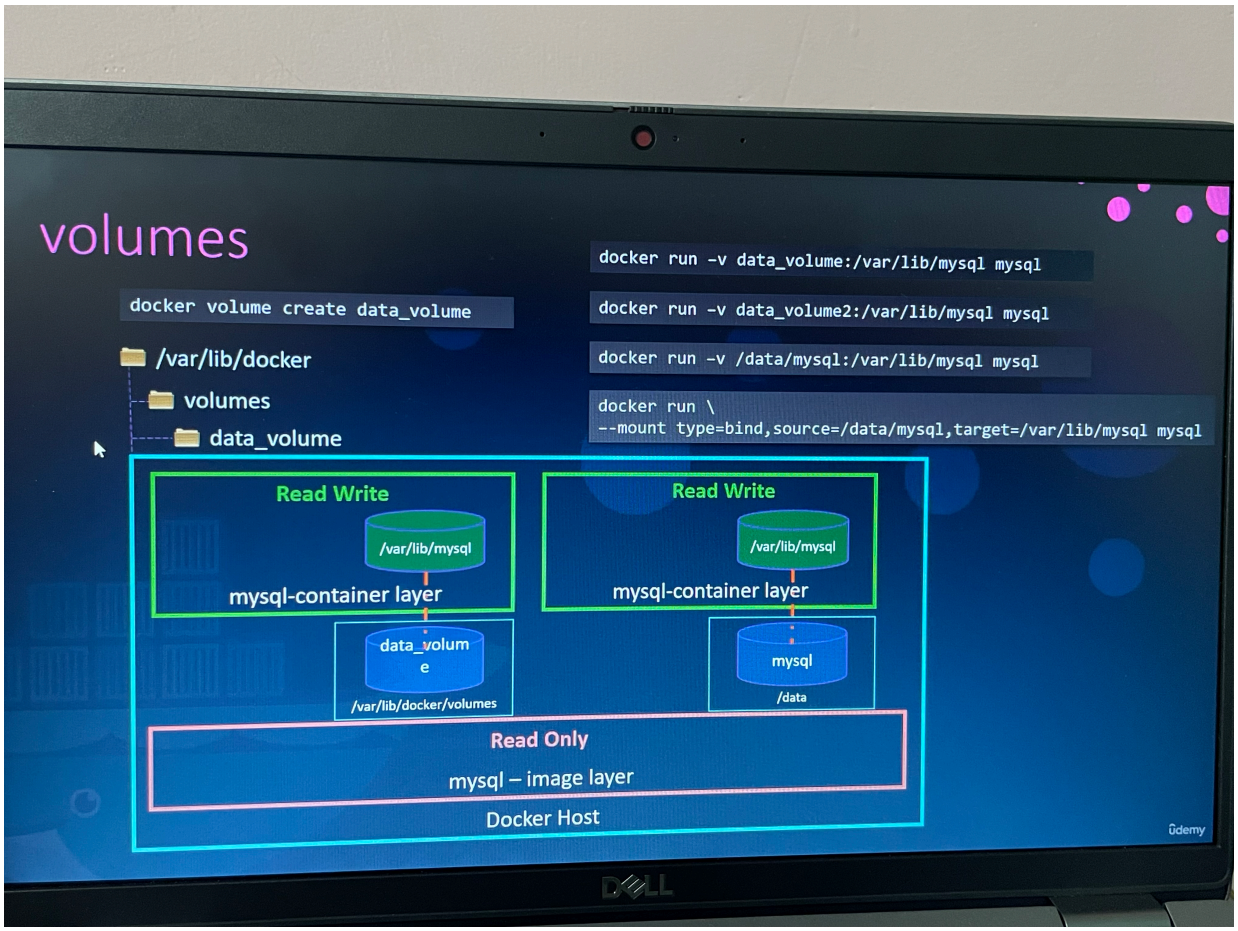
EX: 4 To add egress traffic rule from the db pod to the external backup server on its IP



There are tools like kubectx and kubens to easily switch between contexts and namespaces in a real env easily.

STORAGE:

Default volume location: /var/lib/docker/volumes (this type is called volume mounting). To use a volume in a location other than the /var/lib/docker/volumes, use full path as below. (this type is called bind mounting)



If docker_voulme2 is not created as prerequisite, it gets created when u specify mount command.

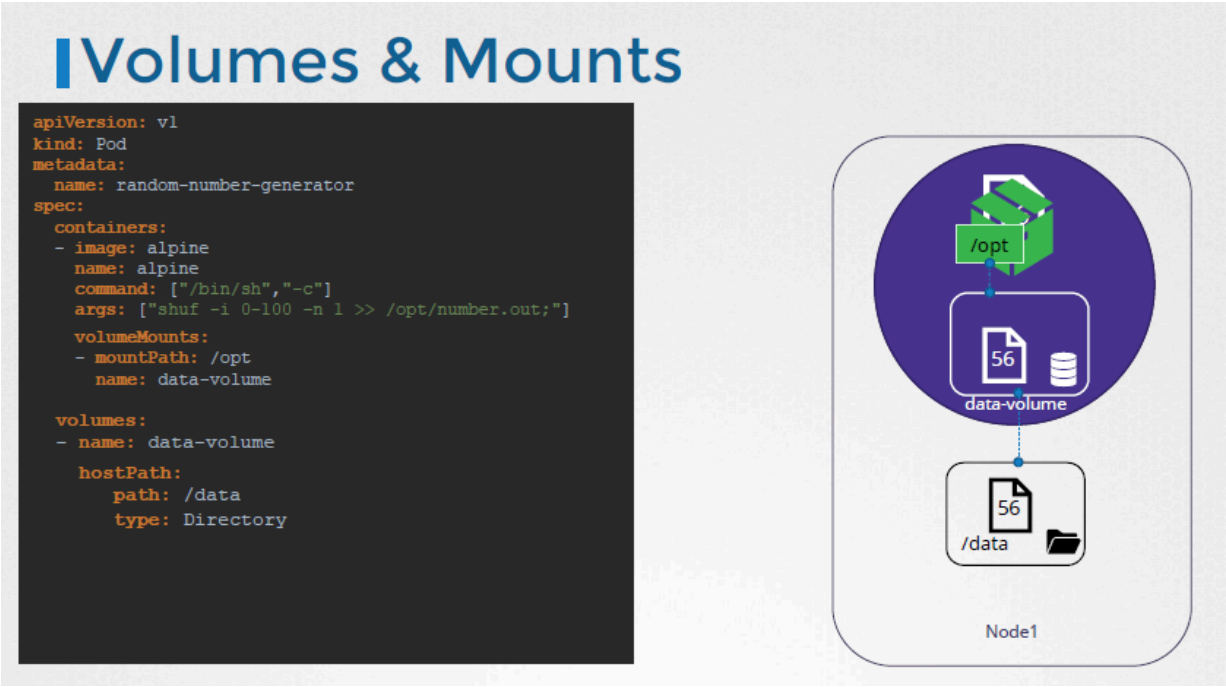
docker run --mount is the latest command for same functionality.

- This volume creation, maintenance is taken care by volume drivers.

CSI:

just like docker (container d), rkt are the CRI - container runtime interfaces, we have couple for CNI - Container network interfaces for networking solution and we also have for CSI - container storage interface like DELL EMC, Amazon EBS which Using CSI, third-party storage providers can write and deploy plugins and expose storage systems in Kubernetes, without touching the core Kubernetes code.

- We have similar method in k8s, where we create volume on node/ cloud storage solution and mount it to the containers.



while this is ok for a single worker node, it is not recommended for a multi node (if there is no data replication) as pod when recreated can go to a diff node from its first node.
BETTER WAY TO MANAGE CLOUD STORAGE SOLUTION FOR VOLUMES: using pv object in k8s we will have to first create a storage in a cloud solution and create corresponding PV object in k8s. (this is called static provisioning)

Now we will have to create persistent volume claim to bind this PV to a application deployment. PV and PVC will have 1-1 relationship meaning no 2 PVC's can consume from single PV though there is space left on a PV.

pvc-definition.yaml

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

- Above PVC will be claimed from the created PV though the requested size is < PV size as there is a single PV available. If we delete PVC, the PV behaviour depends on setting : persistentVolumeReclaimPolicy: Retain/ Delete / Recycle we define on the PV yaml file.
- retain --> is the default; Once the PVC using this PV (with retain policy)is deleted, the PV will remain in the cluster but no longer associated with the PVC. The data on the PV persists and can be reused manually by binding to another PVC or deleted by an admin. This is used when the data needs to be backed up or transferred elsewhere.
  - The PVC will be in terminating state unless the pod(s) using this is deleted
  - once we delete the pod, the PVC is deleted fully and the PV will be in retained status.
- Delete --> Once the PVC is deleted, the PV is also deleted along with the underlying storage object. Used when the storage is not required beyond the lifetime of a PVC.
- Recycle --> data erased and ready for next use. (deprecated in newer releases)

--> Dynamic provisioning using cloud/ external storage solutions: we need to create only storage class, PVC file using that storage class and bind this PVC to pods.
we need to create a STORAGE CLASS manifest file.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gcp-storage
provisioner: kubernetes.io/gce-pd
reclaimPolicy: Retain # default value is Delete for gcp maybe ??
parameters:
```

```
  type: pd-standard  (pd-standard/ pd-ssd)
  replication-type: none (none/ regional-pd)
#The parameters are very specific to the storage provider. There are azure file, azure disk, aws ebs solutions.
```

And then create a PVC which uses this storage class.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: gcp-pvc
spec:
  storageClassName: gcp-storage
  accessmodes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 50Mi
```

Example for storage classes:

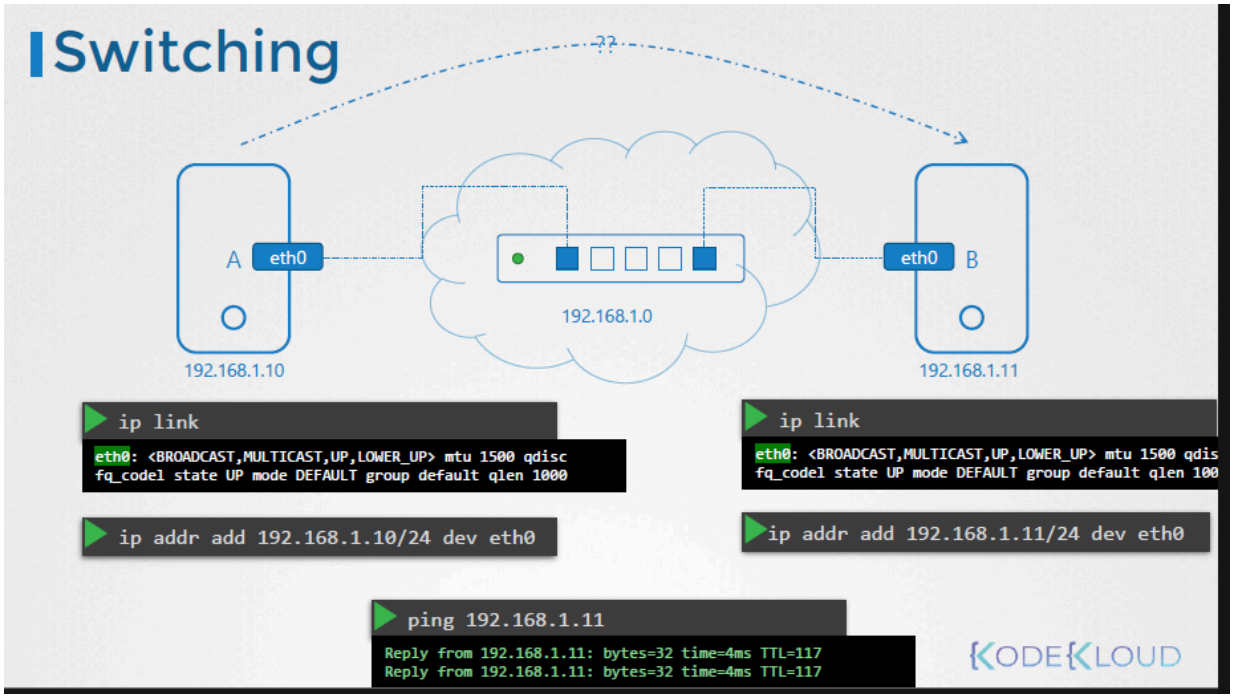
```
controlplane ~ ➜  k get sc
NAME                     PROVISIONER                    RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
delayed-volume-sc        kubernetes.io/no-provisioner   Delete          WaitForFirstConsumer   false                  5s
local-path (default)     rancher.io/local-path          Delete          WaitForFirstConsumer   false                  27m
portworx-io-priority-high   kubernetes.io/portworx-volume   Delete          Immediate              false                  21m
```

** the one with provisioner as no-provisioner means it doesn't support dynamic provisioning.
** If any storage class uses "Volume binding mode = wait for consumer", though we create a pvc, it will wait for any pod that uses this volume and then binds the pvc to the pv.

NETWORKING BASICS

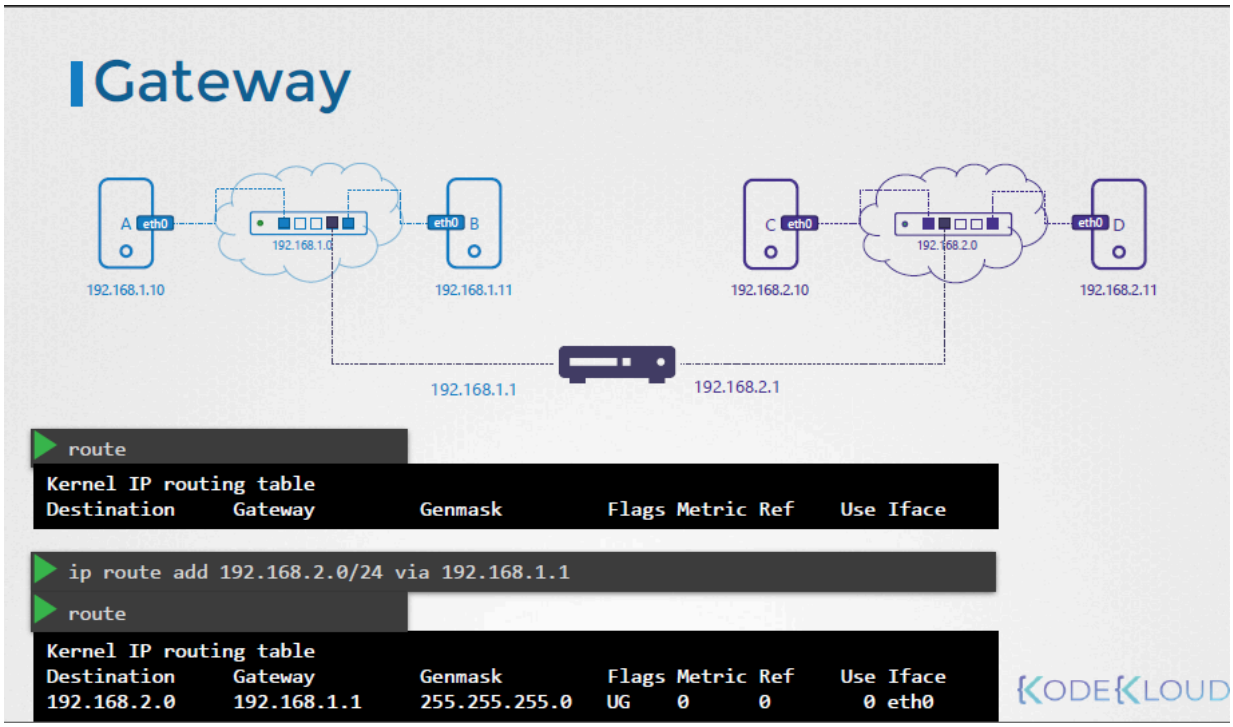
SWITCHING: switch is a device used to connect multiple devices together to form a network. For each device (laptop/system), there should be a interface ("ip link" command) and now

assign one IP for each system from the network. Now the systems can communicate with others which are in the same network.



ROUTING:

Router is a device which acts as gateway to connect multiple devices to the internet/ intranet. i.e., it helps to connect multiple networks together, and can connect multiple networks to the internet.

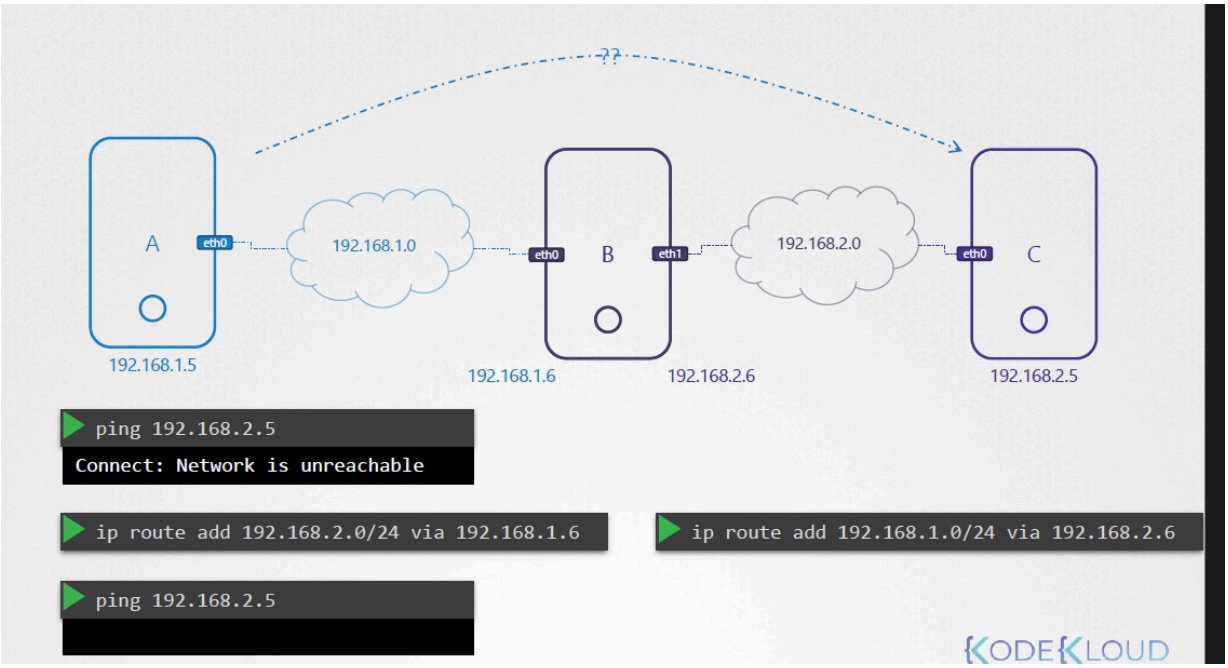


GATEWAY concept: The router gets 2 IP's, one from each n/w. Here the routers IP will be the gateway (for that network) to reach other network.
Above we are adding a route entry in a system in 192.168.1.0 n/w to reach 2.0 n/w via its gateway IP. Suppose these systems need internet, we will connect router to internet and can add google address route but there are multiple sites on internet. we cannot add for each route entry to use this router as gateway. So simply, we add this router IP as default gateway as below:

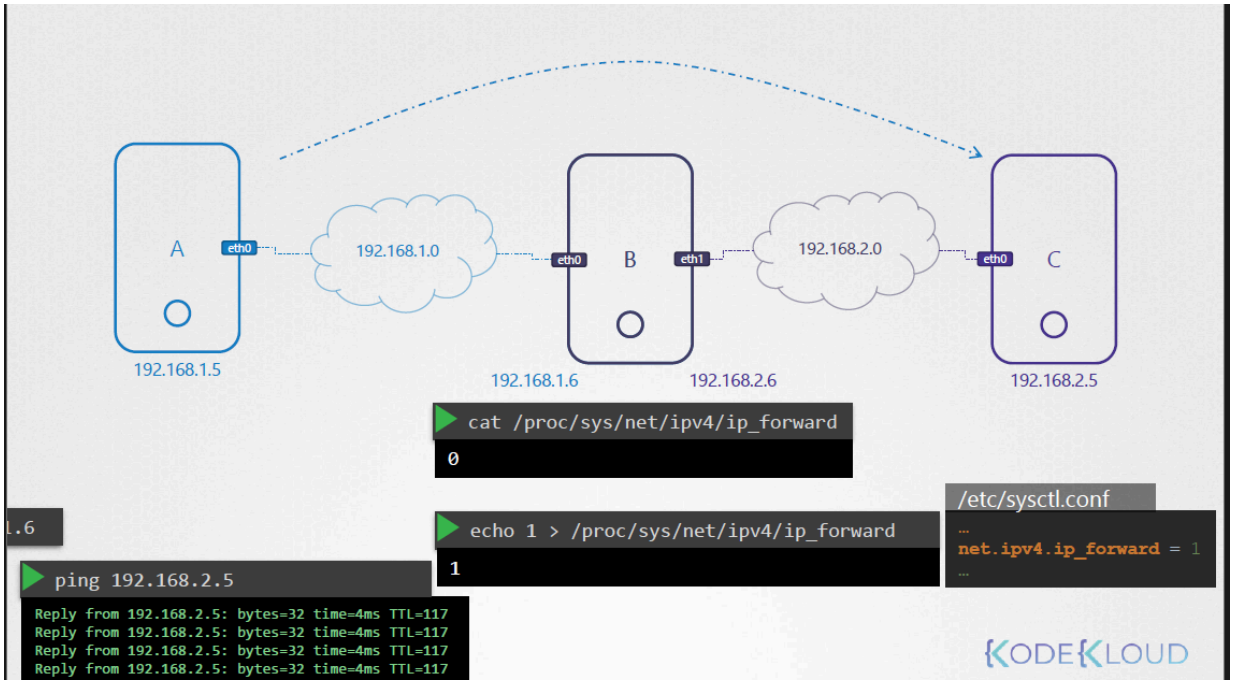
```
ip route add default via 192.168.2.1
```

```
route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.0     192.168.2.1     255.255.255.0   UG    0      0        0 eth0
0.0.0.0         192.168.2.1     255.255.255.0   UG    0      0        0 eth0
192.168.2.0     0.0.0.0         255.255.255.0   UG    0      0        0 eth0
```

here the destination 0.0.0.0 means to everywhere.
gateway 0.0.0.0 means no gateway is needed as the destination is in the same n/w.
CONFIGURING A COMPUTER AS A ROUTER:



If there are 2 private networks and system B is connected to both of the n/ws on 2 of its interfaces, it will get 2 Ips one from each network. A <==> B , B <==> C, but A !<==> C. for A to talk to C, there should be route in between them using B. Now we need to enable packet forwarding in system B for system-A,C to talk to one another using B.

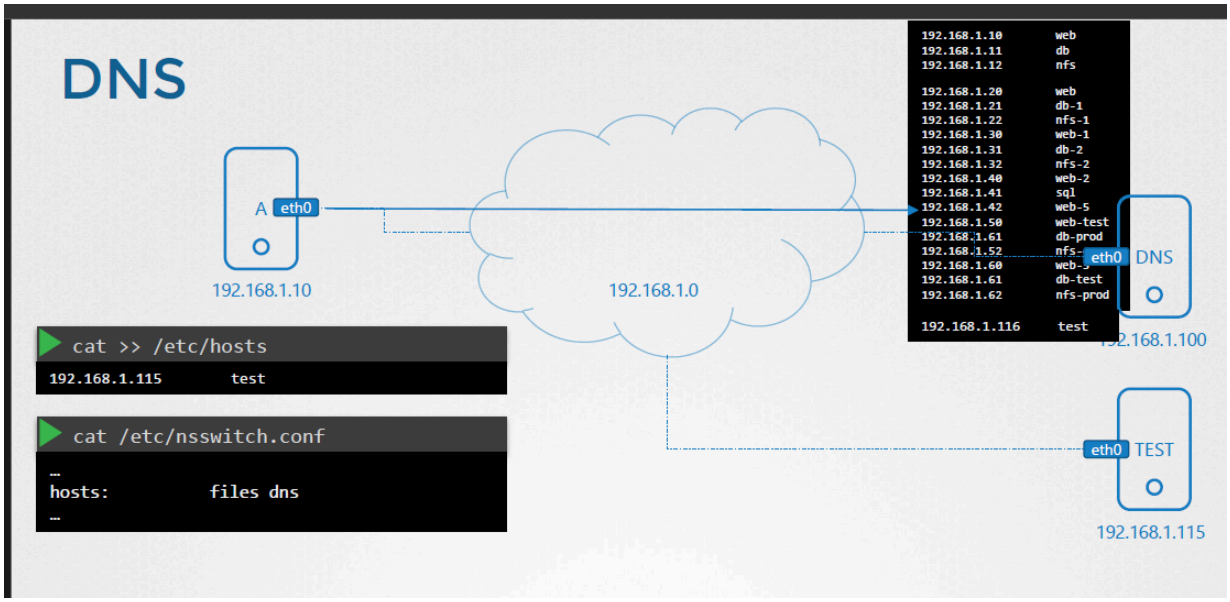


To have the ip_forward setting to be persistent, we will add in /etc/sysctl.conf file.

ip link --> to view the n/w interfaces on the host
ip addr --> to view ip addresses of those interfaces
ip addr add <ip> dev eth0 --> add ip address to the interface
To persist the changes, /etc/networkinterfaces file.
ip route/ route --> view route table
ip route add <ip/ ip CIDR> via <ip>
cat /proc/sys/net/ipv4/ip_forward --> value as 1 to enable packet forwarding.
ip addr show type bridge --> shows interfaces of type bridge
netstat -npl | grep -i scheduler --> shows the ports the kube-scheduler is listening to.
netstat -anp | grep etcd | grep 2379 | wc -l --> to know how many clients are listening on a specific port

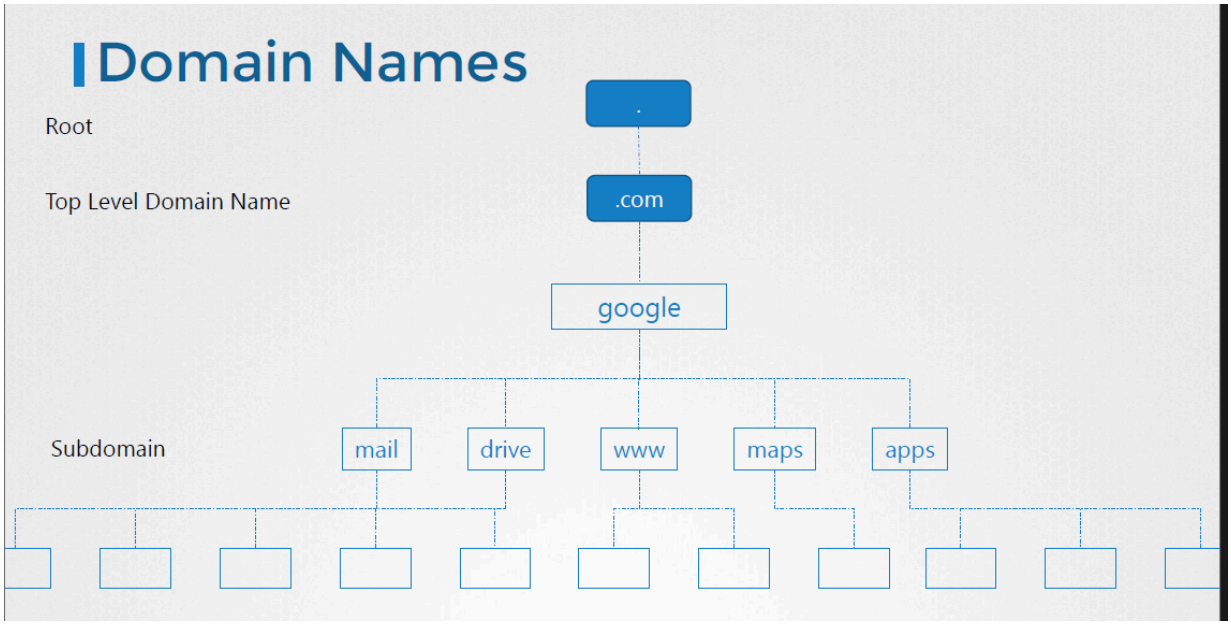
DNS

- In conventional way, for any name resolution, the system looks into its /etc/hosts file for entry (has mapping of IP add to hostname) but it will be hard to manage in above way, we use a DNS server for name resolution. we add the entry like "nameserver <ip of dns server>" in the /etc/resolv.conf
- At first the system looks for its local /etc/hosts file for resolution, if not found , it checks in dns server. Though this order can also be changed as below.



Suppose if there is a private dns server which knows only the organisational dependent DNS but not all public DNS, in this case we can add entry to route ALL to public dns server (8.8.8.8 this is public dns hosted by google). like : "Forward All to 8.8.8.8" on the dns server.

www.google.com --> DOMAIN NAMES example



apart from .com, we have .edu, .org, .in

Above entry used to append full domain names when we specify subdomain like web --> web.mycompany.com, now we can use "ping web" and the system tries to resolve and reach for all the entries as per resolv.conf --> web.mycompany.com/web.prod.mycompany.com

Record Types

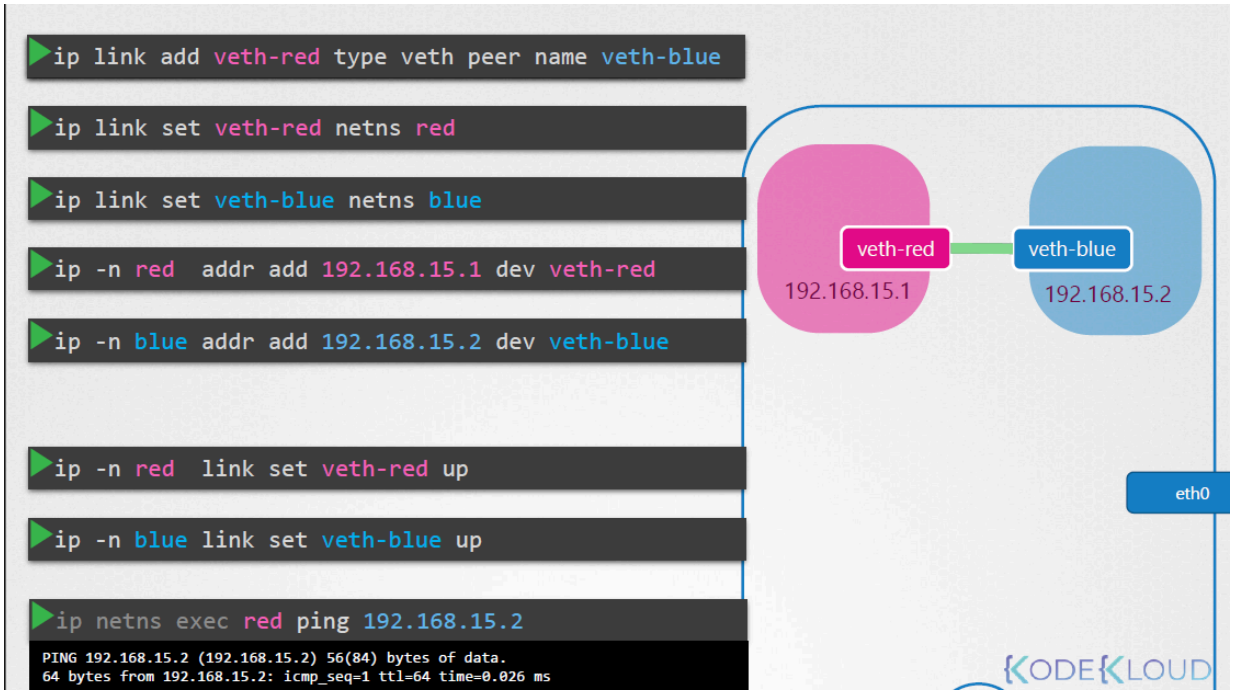
A	web-server	192.168.1.1
AAAA	web-server	2001:0db8:85a3:0000:0000:8a2e:0370:7334
CNAME	food.web-server	eat.web-server, hungry.web-server

nslookup, dig commands doesn't look at all in local /etc/hosts file, they only check for dns servers for resolution.

NETWORK NAMESPACES:

just like how containers and host is logically separated using namespaces, there are network namespaces too. (ip netns add <ns-name> to create a new n/w ns)

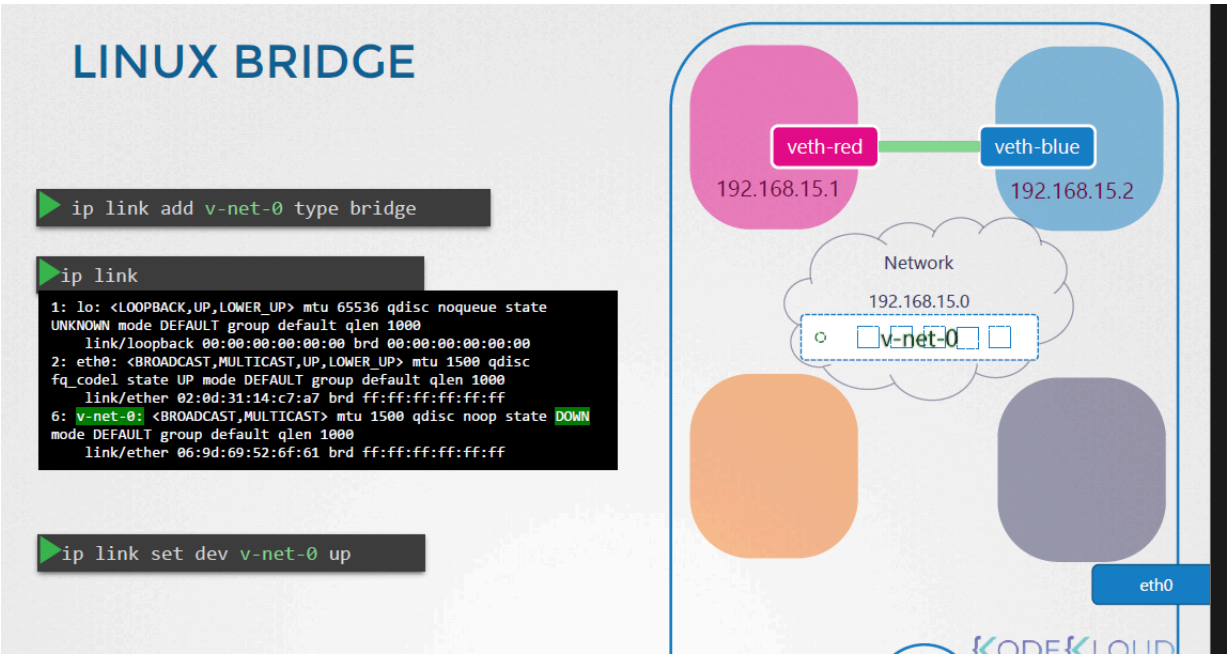
ip netns (to view all the n/w ns), now we need to create virtual network interfaces, associate them to the created n/w ns, and create IPs to the interfaces.



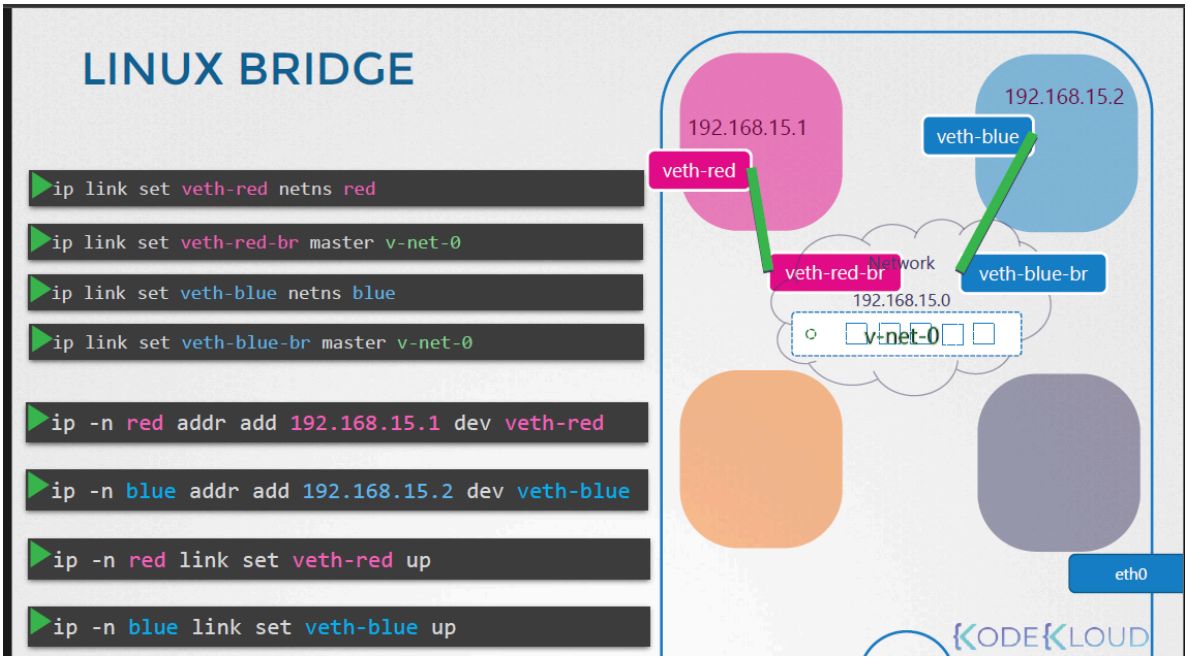
- ip netns exec red ip link / ip -n red link : command to get network interfaces on the red ns.

(same for arp/ route commands)

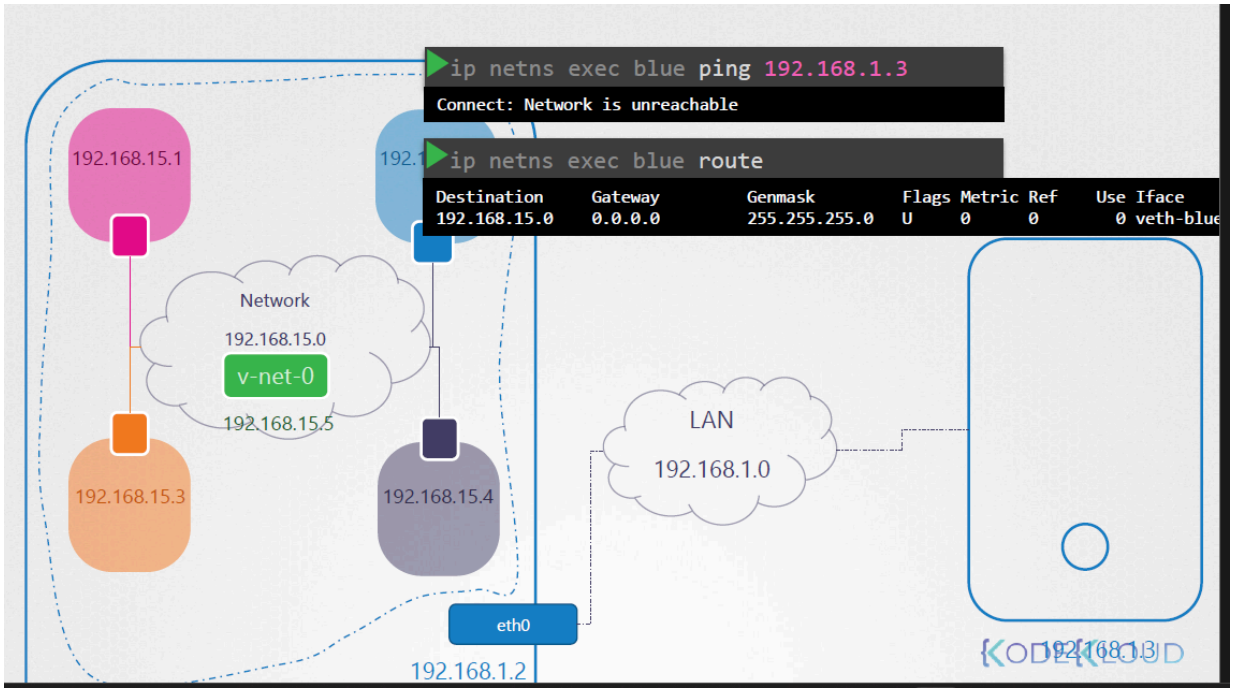
This is example for understanding but in real cases, there will be multiple virtual ns, so we will create a bridge setup.



- we will create a new virtual network interface called v-net-0 (this acts as a switch where multiple systems can connect in real world)
- Delete old cables (ip -n red link del veth-red) and create new ones to connect each namespace to the bridge n/w (ip link add veth-red type veth peer name veth-red-br).
- Now associate one end of the cable to n/w ns and other to the bridge. (ip link set veth-red netns red; ip link set veth-red-br master v-eth-0)
- And add IPs to vnet end of the cable.
-

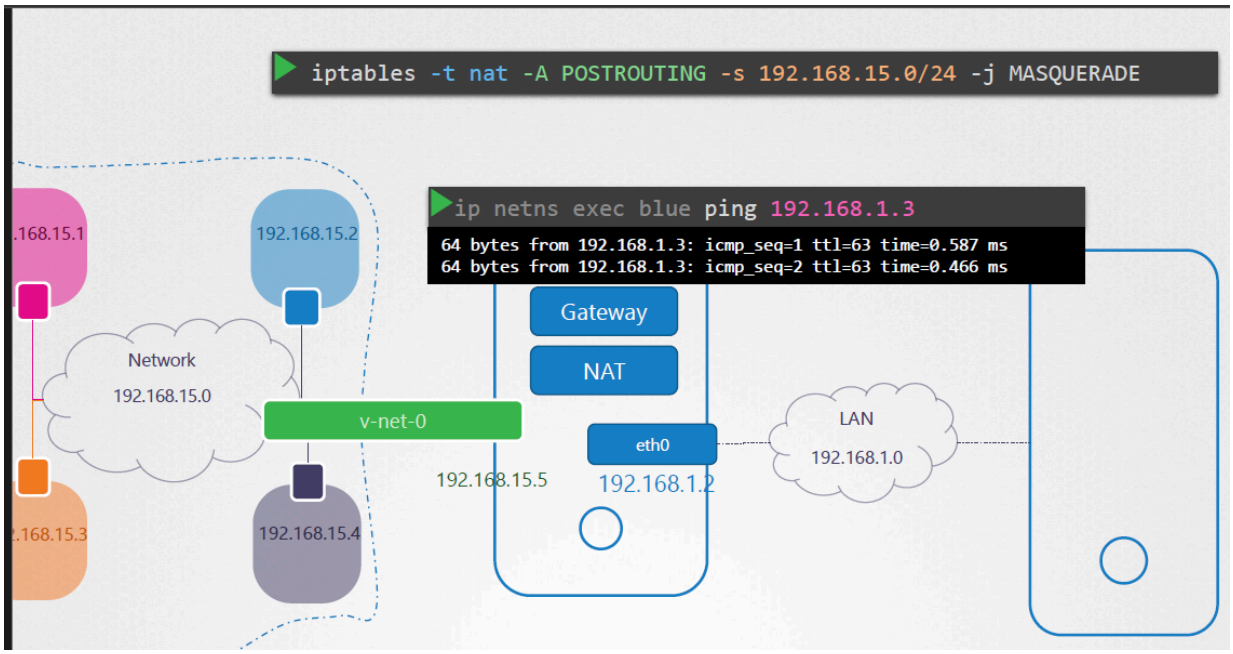


- In same way if done for other ns, all these 4 ns are now in a virtual network connected using bridge.
- However, the host cannot reach this namespaces. to reach from host, assign an IP to the vnet interface created " ip addr add 192.168.15.5/24 dev v-net-0" and now the host will be able to reach the ns. (as acc to the host, this is just another n/w interface and acc to the namespaces, it is a switch)



- These all ns are in a private n/w and can only be reached from host 192.168.1.2. To make them talk to the LAN n/w: (we'll add NAT FUNCTIONALITY TO OUR HOST) our host is the common one in the middle having connection to the private ns n/w and to the LAN too, add

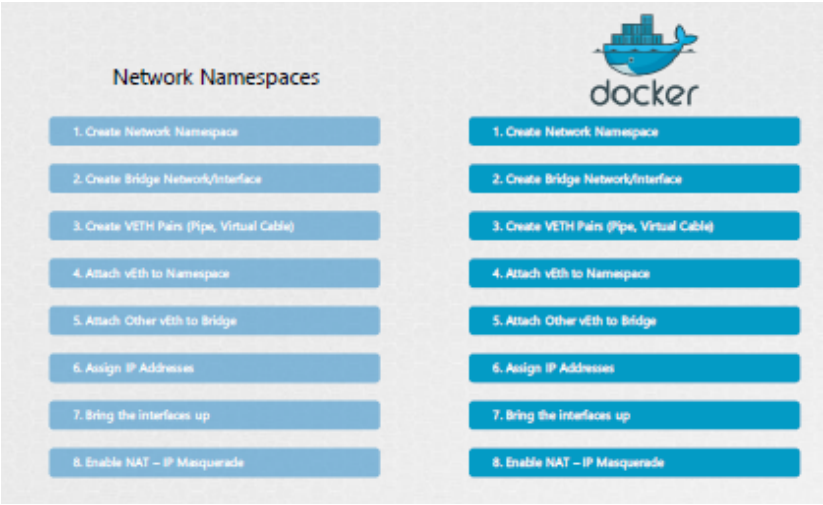
"iptables -t nat -A POSTROUTING -s 192.168.15.0/24 -j MASQUERADE" (this way anyone receiving packets from the pvt n/w will think its being received from the host itself)



- If our host is connected to internet, to enable the ns to reach internet, we will add default gateway as 192.168.15.5
  - ip netns exec blue ip route add default via 192.168.15.5
- To make others to reach the ns's, we will need to do port forwarding on host, whatever is reached on port 80 will be forwarded to blue ns.
  - iptables -t nat -A PREROUTING --dport 80 --to-destination 192.168.15.2:80 -j DNAT

NETWORKING IN DOCKER

Docker also used bridge network as a n/w solution to the containers.
each container ==> each n/w ns as above
each container is joined to the bridge n/w using cables just like we discussed above. and outside world can talk to the containers using port forwarding:
i.e., any traffic reaching on host: 8080 will be forwarded to the container Ip :80.



- So for any Container runtime env/ kubernetes, the networking method is almost the same so, CNI (container n/w interface) came into picture which is set of standards how a n/w solution should work so that these can be used by any CRE's in the form of plugins.

KUBERNETES NETWORKING

once the control plane, worker nodes are ready, each node will need to have a n/w interface connected to the n/w and each interface will get an IP; node will have its hostname and mac address.

- There should be some ports open for the control plane components.

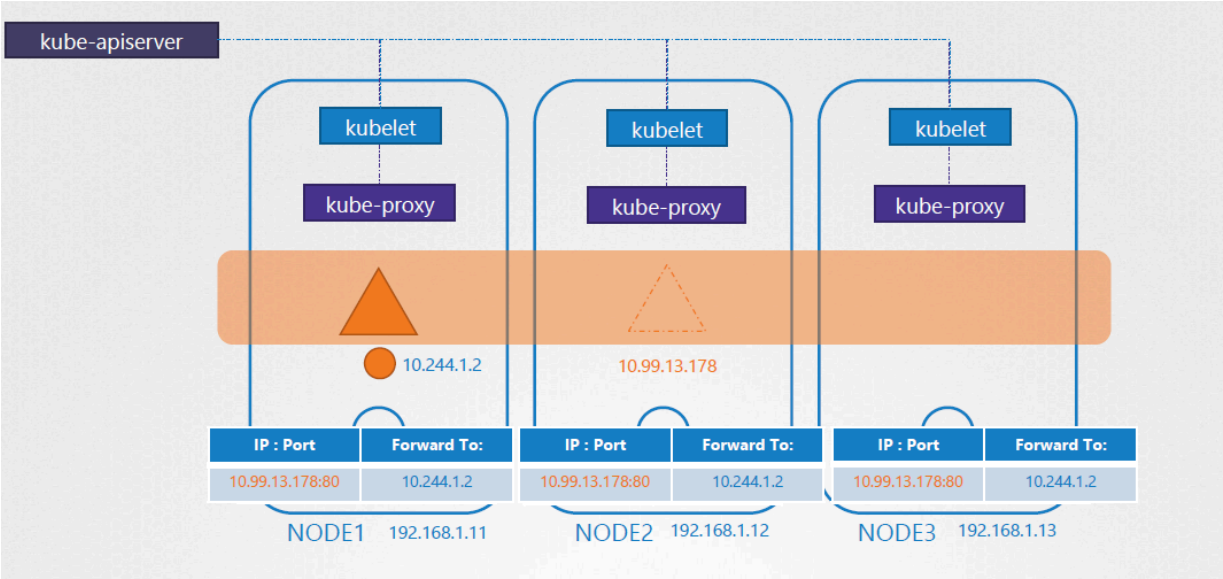
1. 6443 -> apiserver
2. 10250 -> kubeletes on master/ worker
3. 10259 --> scheduler
4. 10257 --> controller manager
5. 30000 to 32767 --> on worker nodes for node ports
6. 2379 --> etcd server ; 2380 --> etcd peer

- /opt/cni/bin/ --> contains binaries of all CNI supported n/w plugins.
- For the pod networking, we use n/w solutions as plugins (ex- weave)which are indeed deployed based on CNI standards. "/etc/cni/net.d" has which solution is in use and its info.

(if there are multiple files, alphabetical order is chosen)
EX: weave is installed as daemon sets i.e., it runs one pod per node.
IP ADDRESS MANAEGEMENT :

The CNI plugin (the n/w solution) itself manages what IP ranges each node gets, there by avoiding duplicate IPs for the pods in the cluster.

- kubelet is the component that creates pods, so whenever a pod is created, kubelet invokes the cni plugin for getting the Ip to the pod and attaching it to the network.
- similarly, whenever a service is created, kube proxy invokes cni to get Ip for the service object. The difference here is pod n/w has network interface, IP assigned to it and all... but the service is just a virtual object. Whenever one pod try to reach other pod using its service name/ IP, kubeproxy maintains a table like below:



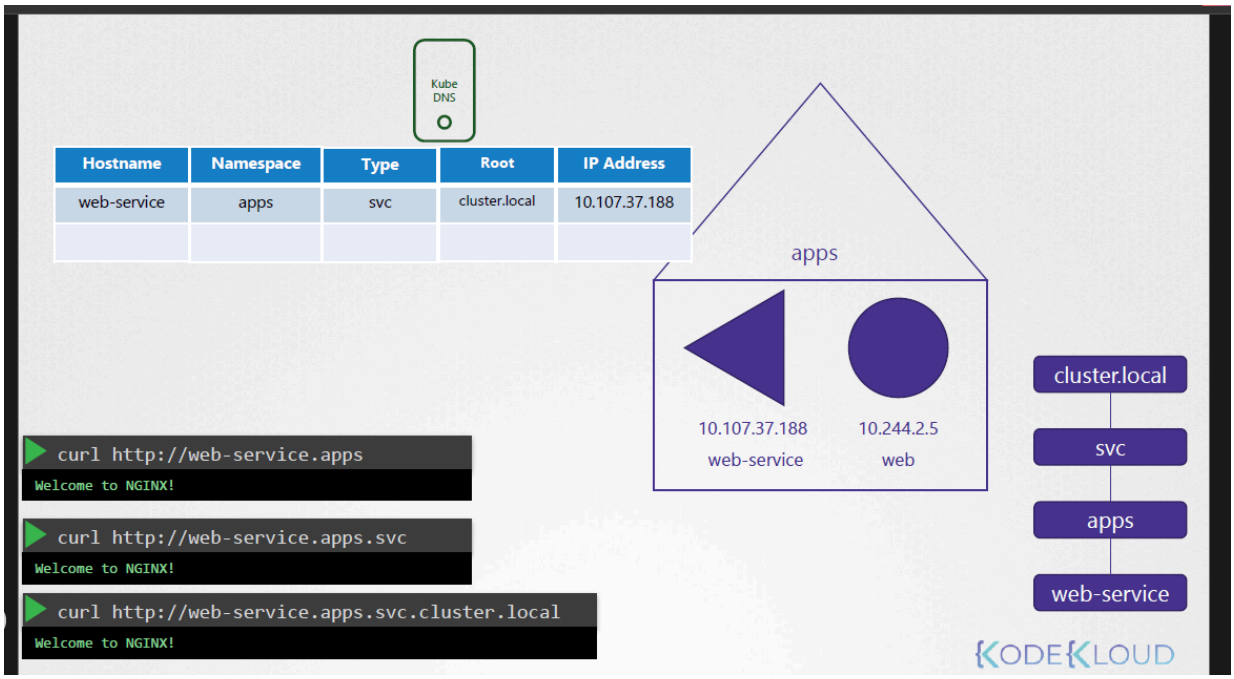
so whenever one pod tries to reach other using service Ip: port(10.99.13.178: 80) it will be forwarded to the backend pod. This IP table is maintained across all the nodes.

- kube proxy will have diff ways to save this info of mapping service IP: port --> pod IP like :
  - kube-proxy --proxy mode [userspace | iptables | ipvs]
- Kube api server will have a ip range for cluster IP (service type- cluster IP) allocation : kube-api-server --service-cluster-ip-range ipNet (Default: 10.0.0.0/24)

this is for division bw the pod IP range and cluster IP range.

To know pod IP range-> check the n/w solution used --> ls /etc/cni/net.d --> or check the pods in kube system for n/w solution. and check the pod logs for ip alloc range.
To know node IP range --> check ip addr in one of node --> check the nodes Ips and check which interface is having this range.
To know IP range for services configured within the cluster --> check for the configs in kubeapiserver pod manifest file/ kube-apiserver process--> "--service-cluster-ip-range"
To know which kind of proxy method kube-proxy is using (ipv, iptables userspace) check the logs of kube-proxy pod.

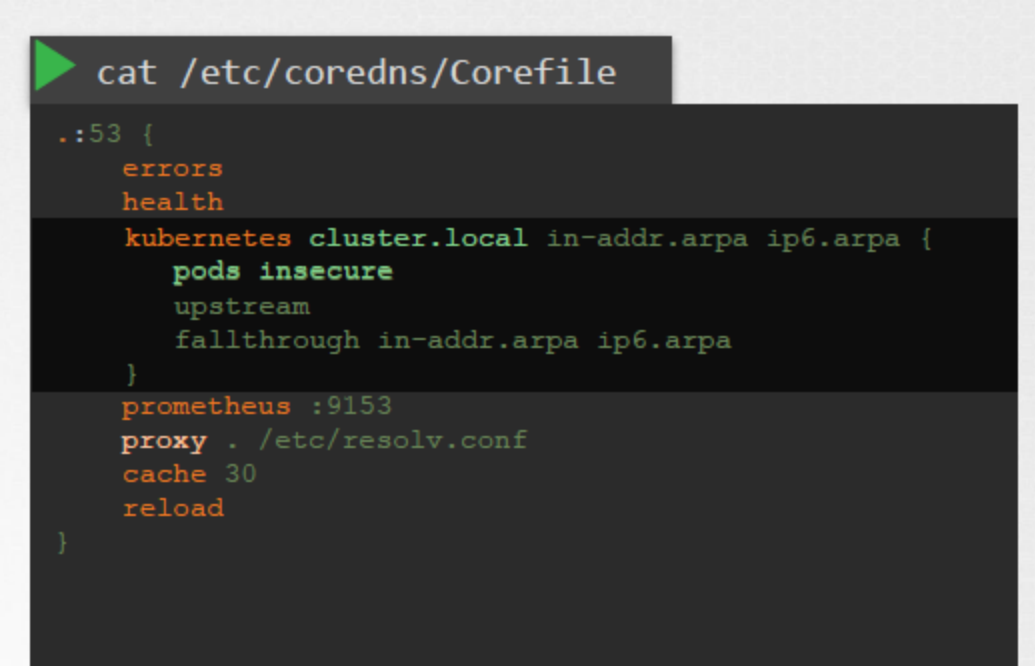
DNS: For a pod in one ns, to reach other pod-service in other ns...we will have to call using its sub domain names.



DNS records for services are created like : cluster.local is the root level domain; all services are grouped in "svc" sub domain; and all services and pods are groups in its own ns named sub domain;
by default pod records aren 't created but when we enable, created as like : 10-244-1-5.default.pod.cluster.local

DNS SETUP IN K8S:

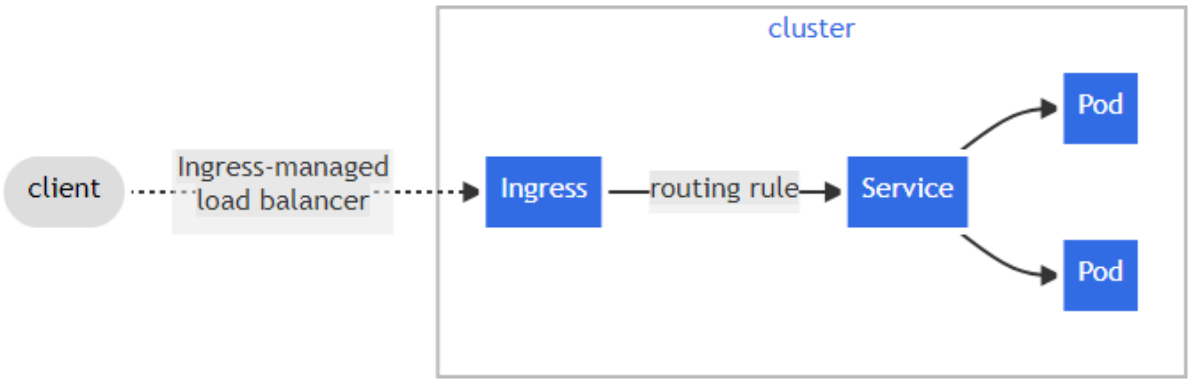
- core dns is the solution used in k8s, it is deployed as deployment : has 2 replicas usually. The core dns solution has the executable & config file : /etc/coredns/Corefile

- 

- cluster.local --> is the root level domain specified; proxy --> /etc/resolv.conf is the file containing "nameserver <mapped to node's nameserver>". if any name resolution cannot be done by the core dns (ex: www.google.com cannot be resolved by coredns solution), it forwards to node's nameserver.
- In each of the pod specification, it has /etc/resolv.conf mapped to "nameserver <core-dns service of type cluster Ip>
- for each pod, in resolv.conf file it also has "search <ns>.svc.cluster.local svc.cluster.local cluster.local" so pods in same ns can access other service using any of "<service-name> alone / <service-name>.ns / <service-name>.ns.svc / FQDN.
- for pods in other ns, we will have to pass atleast <service-name>.ns / FQDN as per above search entries it works like that.
- this search entries are only for services, for pods we will have to pass FQDN.

INGRESS:
The Ingress concept lets you map traffic to different backends based on rules you define via the Kubernetes API. Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.
You must have an Ingress controller to satisfy an Ingress. Only creating an Ingress resource has no effect.
You may need to deploy an Ingress controller such as ingress-nginx. You can choose from a number of Ingress controllers.

Here is a simple example where an Ingress sends all its traffic to one Service:

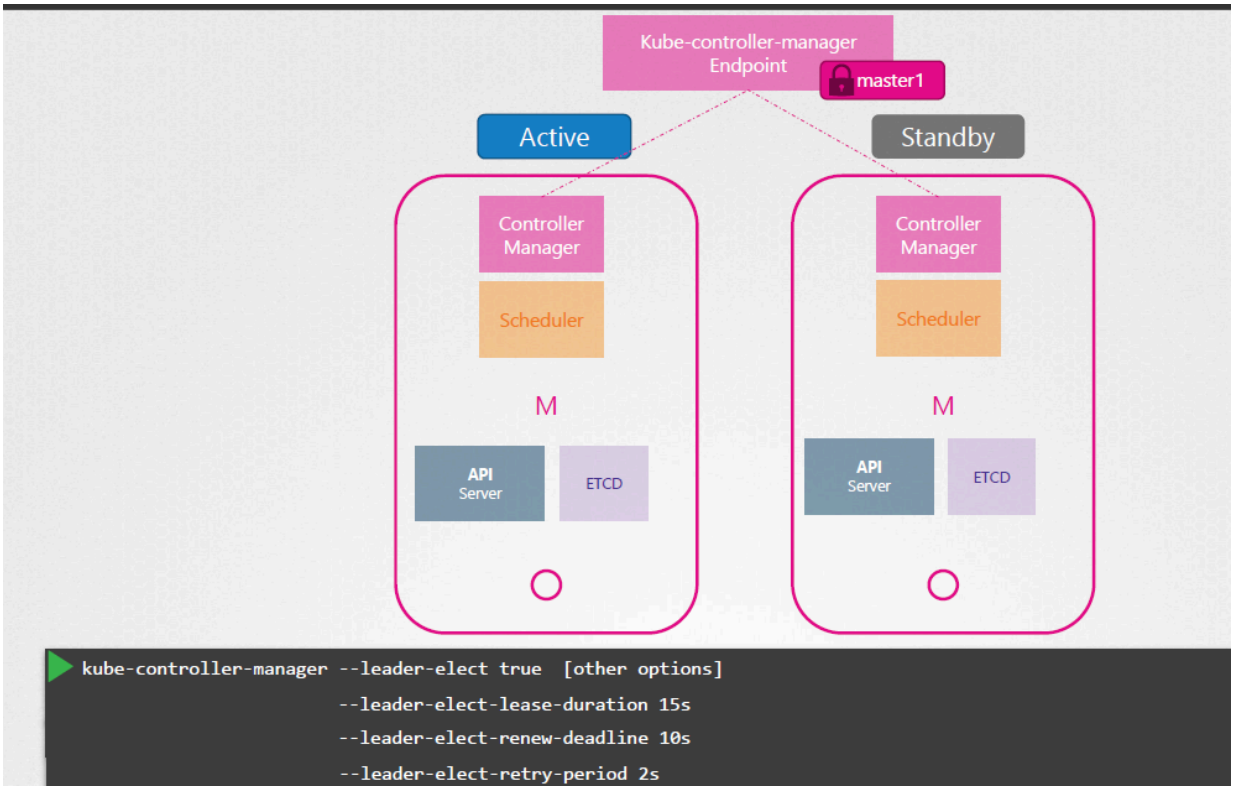



```
apiVersion: networking.k8.io/v1
kind: Ingress
metadata:
  name: ingress-ex
  namespace: critical-space
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /stream
        pathType: Prefix
        backend:
          service:
            name: video-service
            port: 80
```

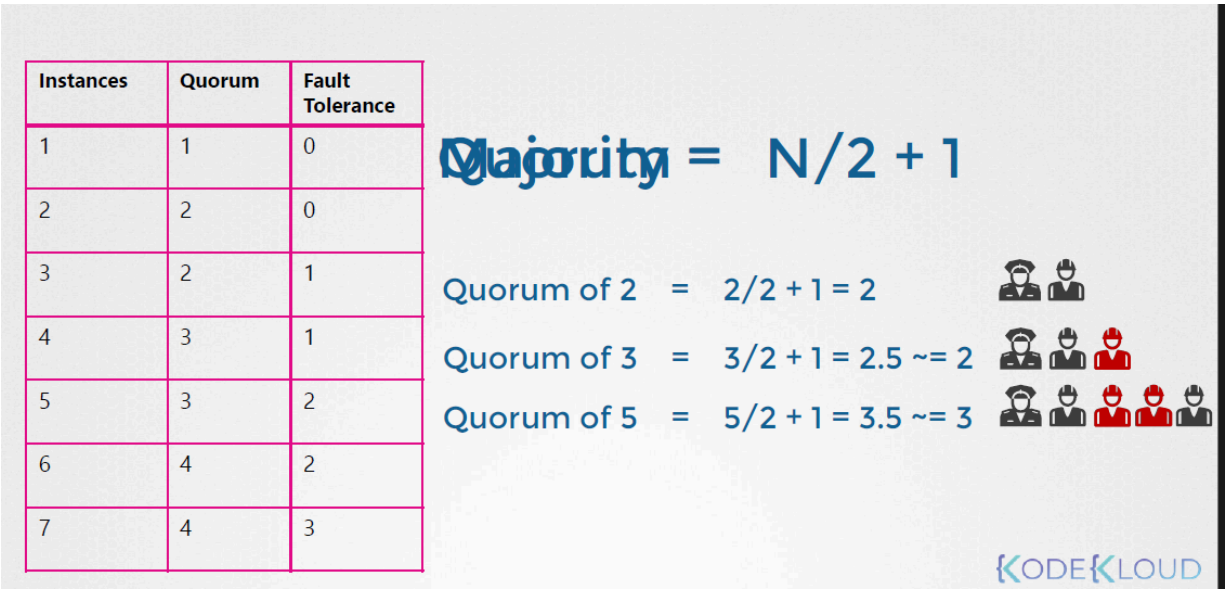
IMPERATIVE COMMAND FOR CREATING INGRESS:
kubectl create ingress ingress-test --namespace=critical-space --rule="/pay=pay-service:8282" : this will not add any annotations.
NOTE: usually the applications are served at "/" so in the ingress rule if we have path based routing, we will need to re write the target to change "/pay --> /" and pass to the backend service.

OTHER POINTS ABOUT K8S SETUP:

- in case of more than 1 masters, we will need to setup a loadbalancing solution to the kube-apiserver. so that traffic to the api server gets splitted and only 1 master answers the calls.
- In the case of controller manager and scheduler only 1 should do the tasks as working in parallel may lead to duplicate pods. One of the controller manager/ scheduler components - master nodes will be active and others will be standby using below setting - first come first elect as active.



- For ETCD, its a distributed DB which only kube apiserver will talk to. We can have stacked etcd (having etcd running on same master cluster) or external etcd. But in both ways, we will specify etcd endpoints list in the kube api server config. As its a distributed db, api server can read/ write to any in the list.
- How ETCD ensures data consistency ?? --> for multiple nodes, leader will be elected (using RAFT algorithm) and the other follower nodes sends all the "write" requests to the leader. Leader will send copy of that data to all the followers.
  - Followers will send heartbeats to ensure leader is alive, if not found alive; followers will start the process of electing new leader within themselves.
- Concept of QUORUM:



- QUORUM -> the no of etcd nodes required to ensure cluster works properly and that we can say the write operation is completed only when data replicated to the followers as per quorum number. i.e., If we have 3 total nodes, for a write operation to say successful, we should have at least 2 total active nodes; 1/3 can fail at max --> fault tolerance=1.

Having ODD no.of managers is recommended. and Min no =3 for HA Etcd setup.

SETTING UP K8S USING KUBEADM:

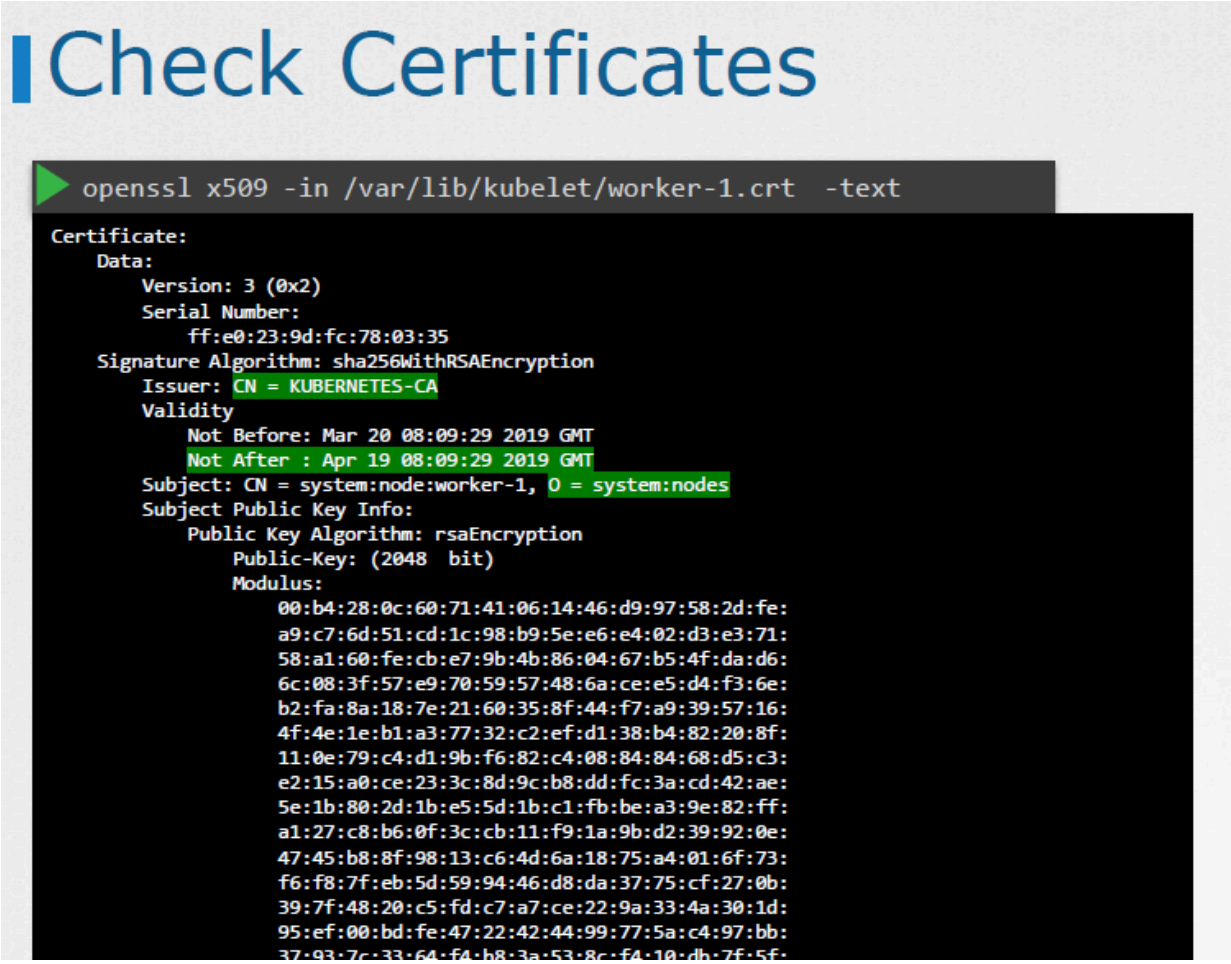
1. Install virtual box, vagrant and create 3 nodes with the required specifications. 1-Master and 2 - Workers.
2. Install kubeadm (kubelet and kubectl), container runtime, C- groups (to manage resource limits on containers) Its imp to make sure both container runtime and kubelet use the same c-groups and are configured to use the same.
   a. ps -p 1 ---> shows the default init system used on our linux node. In this case, use systemd as the c-group driver
   b. For kubernetes version 1.22 or later, the default c-group is systemd. If u want to change this we'll need to create a kuber config yaml file and pass during kubeadm init. "kubeadm init --config kubeadm-config.yaml"
   c. In recent versions of kubeadm, systemd is the default one. Check the references for setting c-group and restart containerd.
3. kubeadm init by passing the required pod network CIDR, controlplane IP address info and certs required info for kubeadm to do all the required setup.
4. Deploy a n/w plugin; it deploys using default pod n/w ; if u wanna change, edit in the file and apply it to kubectl.

5. connect worker nodes to the master using the command we got from above command - 3.

TROUBLESHOOTING:

--> kubectl config set-context --current --namespace=develop

- service kube-apiserver status --> to check status of services when control plane components are running as services;
  kubectl get pods -n kube-system --> when components are running as pods
- sudo journalctl -u kube-apiserver --> to check logs of component
  kubectl logs kube-apiserver -n kube-system
- use top/ df -h --> commands to check the memory, cpu and disk status of worker nodes.
- service kubelet status --> to check status of kubelet when running as service.
- sudo journalctl -u kubelet --> to check logs of kubelet.
- Also check certificates --> issuer/ expiry / group allocation for kubelet.



JSON PATH

- $ --> is the root dictionary / list
- In a list like : [12, 14, 45, 46, 43, 55, 45, 90, 100] to get numbers which are > 40:
  $[?(@ > 40)] there are other operators like : @ == 40
  @ != 40
  @ in [43,45,47] / nin for not in
- k8s use case:
  - kubectl config view --kubeconfig=my-kube-config -o jsonpath="{.contexts[?(@.context.user=='aws-user')].name}" > /opt/outputs/aws-context-name

```
[
   "car",
   "bus",
   "truck",
   "bike"
]   --> cat file.json | jpath $[1]  --> [ "bus" ] as output.
cat q13.json | jpath '$[0,3]'  --> ["car", "bike"]  as output.

{
   "property1": "value1",
   "property2": "value2"
}
--> cat q1.json | jpath $.*
  gives:
[
  "value1",
  "value2"
]

[
   {
      "model": "KDJ39848T",
      "location": "front-right"
   },
   {
      "model": "MDJ39485DK",
```

```
        "location": "front-left"
    },
    {
        "model": "KCMDD3435K",
        "location": "rear-right"
    },
    {
        "model": "JJDH34234KK",
        "location": "rear-left"
    }
]
--> cat q4.json | jpath '$[*].model'
[
  "KDJ39848T",
  "MDJ39485DK",
  "KCMDD3435K",
  "JJDH34234KK"
]

[
"apple",
"mango",
"banana",
"grapes",
"pineapple",
"lemon",
"orange"
]
--> cat file.json | jpath '$[0:3]'   will give apple,mango and banana note: 3 doesn't give upto
4th element but it gives 0,1,2 elements
--> cat file.json | jpath '$[0:7:2]'  will give alternate values from 0 till end.   [START:END:STEP]
-->  $[-1:] to get last element
--> $[-3:] to get last 3 elements.

[
  "Apple",
  "Google",
  "Microsoft",
```

```
  "Amazon",
  "Facebook",
  "Coca-Cola",
  "Samsung",
  "Disney",
  "Toyota",
  "McDonald's"
]
-->cat input.json | jpath '$[0:5:4]'
[
  "Apple",
  "Facebook"
]
--> cat input.json | jpath '$[-7:-1]'
[
  "Amazon",
  "Facebook",
  "Coca-Cola",
  "Samsung",
  "Disney",
  "Toyota"
]
--> cat q8.json | jpath $.prizes[?(@.year == 2014)].laureates[*].firstname
[
  "Kailash",
  "Malala"
]
```

JSON PATH IN K8S:

Kubectl utility fetches the results from kube apiserver and prints in readable format but the actual output is json. we can use json path to extract any data of our choice.

kubectl get nodes -o json --> to view json output, and then u need to build the jsonpath query and add to the command as below.

EX: kubectl get pods -o=jsonpath='{.items[0].spec.containers[0].image}' -> to get the image of a first pod.

we can use * to iterate through each item. and use {"\n"} and {"\t"} for new line and tab.

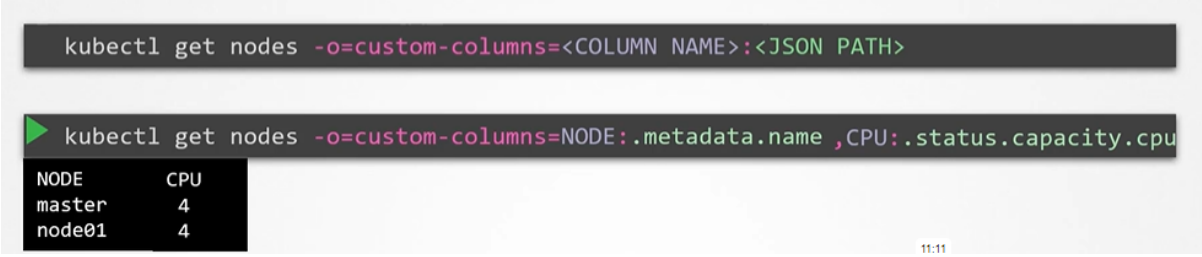
EX: kubectl get pods -o=jsonpath='{.items[*].spec.metdata.name}{"\n"} {.items[*].status.capacity.cpu}' we can also have loops in the command to get details of each

item from the command more beautiful way like the actual kubectl utility does.

```
FOR EACH NODE

  PRINT NODE NAME \t PRINT CPU COUNT \n

END FOR
```

```
'{range .items[*]}

  {.metadata.name} {"\t"} {.status.capacity.cpu}{"\n"}

{end}'
```

easy way is to use custom columns option and specify as below (note: this iterates through each item by default so no need to specify items[*])



we can also use --sort-by to sort out the output wrt to a particular property.