**CKA Exam**

## Kubernetes Core Concepts:

# Cluster Architecture:

1. etcd: works like a database and stores in key:value format.stores information about different nodes, pods, logs etc.

2.Control manager: contains replica set or replica-controller and node controller. it ensures that the required amount of pods / applications are running on the nodes.

3.api-server: main brain which manages all the other components of Kubernetes like etcd, kube-proxy etc

4.kube-scheduler: schedule containers on right nodes based on their availability, resources and other conditions and ensure balance between nodes.

5.kubelet: acts as an agent on worker nodes and manages pods and worker nodes and gives info to kube-api.

6.kube-proxy: for communication between various services on nodes, pods communication etc.

7. Container run-time Environment : it is the underlying infrastructure in which your application runs on eg Docker.

# ETCD:

* It saves data /record in a document or a page so that each record can be updated without affecting other.
* When too many records are there we need to transact into these using YAML or JSON.
* It listens on 2379.
* In Kubernetes, you can configure it manually or by kubeadm.
* It maintains info about every change which get updated in etcd databases, every configuration info etc. and when firing a command all info you get from this database.
* With kubeadm it executes as a pod in the kube-system namespace.
* In Kubernetes it stores these keys in /root registry and then /pods or/deployments etc.

# Kube api-server:

* primary management component in kubernetes.
* whenever a change or command is fired these steps followed generally:
* eg when a pod is created:

--User is authenticated

--Request is validated

--data is retrieved

--etcd is updated

--scheduler schedules pods

-- kubelet then retrieves the image and then runs the image in the container.

* Here kube-apiserver is the center of all the tasks performed .
* Every component 1st ask the api server to do a task and then performs it and only kube-apiserver deals with etcd database.

# Kube-controller

* It is responsible for monitoring the whole system and taking appropriate action if it finds something inconsistent.
* There are Node controller , replication controller ,deployment controller etc

e.g. node controller checks status of Nodes every 5 seconds through api-server.

# kube-scheduler:

* Which pod goes to which node based on the condition of nodes like resources etc.
* It actually does not execute the pod to run; rather it tells this to kublet which is running in that node.
* You can run a multi-scheduler environment and also configure your own custom scheduler.

# kubelet:

* Acts as a captain of worker nodes and a single point of communication with worker nodes.
* Also monitors logs etc all the pods running in the worker nodes and tells it periodically to master nodes.
* Kubelet registers the node in master and creates pods using container runtime which pulls an image.
* kubeadm creates all components of kubernetes in the kube-system namespace (in master-node) but it does not create a kubelet.
* kubeadm actually deploys all control-plane components as pods in master-node though kubelet in a kube-system namespace.

# kube-proxy:

* Pod communicates with each other through a private network. We do this by creating a service in order to communicate.
* Creates a service exposing its port, service has its own ip address. it forwards the traffic to backend pods.
* Service here is a virtual thing or concept it doesn't have an interface or anything (therefore it doesn't have a network or takes part in pod private network). How do kubernetes achieve communication then?
* That's when Kube-proxy comes into play. Every time a new service is created, kube proxy adds appropriate rules for traffic forwarding to backend pods.

# Pods:

* Smallest object unit in kubernetes which encapsulates a container.
* Can have multiple containers in a pod (containers is an array).
* Commands:-
* kubectl get pods
* kubectl logs <pod-name> -c <container-name \*If multiple-container>
* kubectl logs <pod-name> -l <key-name>=<value-name>
* kubectl logs <pod-name> -f <\*stream the logs>
* kubectl run <pod-name> --image=<image-name:version \*creates a deployment to run pod>
* kubectl exec <pod-name> -c <container-name \*If multiple-container>
* kubectl exec -it shell-demo -- /bin/bash or /sh (running interactive shell in a pod, ‘--’ is used to separate kubectl argument & argument used in shell)

-YAML format:

apiVersion: v1

kind: Pod

metadata:

name: nginx-demo

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

# ReplicaSet:

* Used to monitor the pod and if the pod failed to bring up the pod.
* Continuously this process monitors pods and always maintains the number of replicas of pods running.
* If a pod matching a Replica Set is deleted, a new pod with the same configuration automatically sets up.
* You can remove Pods from a ReplicaSet by changing their labels. This technique may be used to remove Pods from service for debugging, data recovery, etc. Pods that are removed in this way will be replaced automatically ( assuming that the number of replicas is not also changed).
* While you can create bare Pods with no problems, it is strongly recommended to make sure that the bare Pods do not have labels which match the selector of one of your ReplicaSets. The reason for this is because a ReplicaSet is not limited to owning Pods specified by its template-- it can acquire other Pods by selector property.

YAML format:-

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: frontend

labels:

app: guestbook

tier: frontend

spec:

*# modify replicas according to your case*

replicas: 3

selector:

matchLabels:

tier: frontend

template:

metadata:

labels:

tier: frontend

spec:

containers:

- name: php-redis

image: gcr.io/google\_samples/gb-frontend:v3

-Commands:-

* kubectl get rs
* kubectl scale rs <rs-name> --replicas=3 (this will affect only live-object with that name not the definition file)
* kubectl replace -f <rs-definition-file-name> (first update parameters like replicas etc. then use command)
* kubectl scale --replicas=6 -f <rs-definition-file-name> (this will update replicas without opening the definition file)

# Deployments :

* Rolling Update:- if you upgraded your app and need to be updated in all the pods and containers then you do it one by one not all of them simultaneously. This one by one approach is a rolling update.
* So in case if the upgrade fails we Roll back to the previous one without any downtime!
* To upgrade the underlying changes or undo changes or resume changes as required we have a Deployment object in kubernetes.

-YAML format:-

apiVersion: apps/v1 *# for versions before 1.9.0 use apps/v1beta2*

kind: Deployment

metadata:

name: nginx-deployment

spec:

selector:

matchLabels:

app: nginx

replicas: 2 *# tells deployment to run 2 pods matching the template*

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

-commands:-

* kubectl create deployment --image=nginx nginx (Create a deployment)
* kubectl create deployment --image=nginx nginx --dry-run -o yaml (Generate Deployment YAML file (-o yaml). Don't create it(--dry-run))
* kubectl scale deployment frontend-deployment --replicas=3 (scale a deployment)
* kubectl edit deployment (most parameters in deployments can be edited not the case with pods.)

# Namespaces:

* A working env with its own set of resources, pods,deployments networks ...so far we are working in a namespace called 'Default-namespace'
* To Isolate properly and also added security kubernetes has also its own namespace for dns service,networking etc called as 'kube-system'
* A 3rd namespace also available to all users where resources are made available.
* e.g. if I want to use the same cluster for both development and production but at the same time isolate the resource between them we can make a namespace.
* Each namespace has their own set of policies that define who can do what?..
* Within their own namespace resources can call each other only by their name or service hostname.
* To call other services outside the name same append the name of that namespace 1st.
* If you want to use Environment variables for applications of different namespaces always remember to append ‘.<namespace-name> ’ in .spec.container section.
* Commands:-
* kubectl get namespaces or ns
* kubectl get <object-name> -n=<namespace-name>
* kubectl create namespace <namespace-name>
* kubectl create -f <namespace-definition-file>
* kubectl describe namespaces
* kubectl delete namespaces <insert-some-namespace-name> (this will delete all the object in that namespace)
* kubectl config set context $(kubectl config current-context) --namespace=<namespace-name> (in order to switch to other namespace)
* If you want to create an object using a definition file pass a ‘namespace=<namespace-name>’ in $.metadata !

-Yaml format:-

apiVersion: v1

kind: Namespace

metadata:

name: <insert-namespace-name-here>

* to limit resources and cpu usage and other metrics for a namespace use a ResourceQuota object and create its YAML file.

-YAML format:-

apiVersion: v1

kind: ResourceQuota

metadata:

name: mem-cpu-demo

spec:

hard:

requests.cpu: "1"

requests.memory: 1Gi

limits.cpu: "2"

limits.memory: 2Gi

# Services:

* Helps in communication b/w pods, external user, internal and external network.
* The Service abstraction enables decoupling of different parts of your software by their functionality, or by any means e.g. front-end & backend tier.
* Types of services:-
* Node Port service: listen to a port on node and then forward it to a port which may contain multiple pods. It makes internal pods accessible on nodes to external users.

-port on the node is referred as 'nodePort', port on the service is referred as 'port' and port on the pod is referred as 'targetPort'

-Nordport range: 30000-32767

* ClusterIP:def1 this service creates a virtual IP inside the cluster to enable communication b/w such as a set of front end servers to a set backend servers.

def2:-Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType.

def3:-It is a basic internal service in which one group of pods communicate to another group of pods through a single point of contact.

* Load Balancer: It provinces a load balance interface which is then used by cloud providers.

Exposes the Service externally using a cloud provider’s load balancer.

* The service finds similar pods through selectors as we have multiple instances of the same application running on a lot of similar pods so after selecting these pods, the service automatically distributes the load from external traffic between them.
* The same thing happens if pods are running in multiple nodes service automatically expands in all nodes and distributes load.
* Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS. The former works out of the box while the latter requires the CoreDNS cluster addon.

-YAML format:

apiVersion: v1 # *If type is not specified than by default it is ClusterIP*

kind: Service

metadata:

name: my-service

spec:

selector:

app: MyApp

ports:

- protocol: TCP

port: 80

targetPort: 9376

This specification creates a new Service object named “my-service”, which targets TCP

port 9376 on any Pod with the app=MyApp label.

for nodePort specify a ‘type’ field:

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

type: NodePort

selector:

app: MyApp

ports:

*# By default and for convenience, the `targetPort` is set to the same value as the `port` field.*

- port: 80

targetPort: 80

*# Optional field*

*# By default and for convenience, the Kubernetes control plane will allocate a port from a range (default: 30000-32767)*

nodePort: 30007

for LoadBalancer:

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: MyApp

ports:

- protocol: TCP

port: 80

targetPort: 9376

type: LoadBalancer

commands:-

* kubectl get service -l <label-name to find this type of service>
* kubectl expose pod redis --port=6379 --name redis-service --dry-run -o yaml (Create a Service named redis-service of type ClusterIP to expose pod redis on port 6379);(This will automatically use the pod's labels as selectors)
* kubectl create service clusterip redis --tcp=6379:6379 --dry-run -o yaml (This will not use the pods labels as selectors, instead it will assume selectors as app=redis. You cannot pass in selectors as an option. So it does not work very well if your pod has a different label set. So generate the file and modify the selectors before creating the service)
* kubectl create service nginx --tcp=80:80 --node-port=30080 --dry-run -o yaml (This will not use the pods labels as selectors...)

## Scheduling:

# Manual scheduling:

* Don't want to rely on a built-in scheduler instead want to schedule it yourself.
* The scheduler looks for the 'nodeName' parameter in the spec section of your pod definition YAML file and if no value is given to the nodeName key then it assigns automatically.
* If there is NO Scheduler then pods are always in 'pending' state.
* You can manually set key -value of nodeName: node01 so that after creating the pod YAML file pod automatically runs in the Node mentioning this works even if there is no scheduler.
* If you want to schedule post the creation of pod then kubernetes won’t allow it, api-server adds a Binding object to the pod & node after the pod is created and in order to schedule the running pod in another node you need to send curl request to api-server’s binding api with Binding object in which nodename is specified..

# Labels and selectors:

* To group things together we need labels and selectors.
* Labels are properties attached to each item.
* Selectors help to filter times based on these labels.
* Kubernetes uses these labels to connect different objects.
* Annotations are also used to record other information like build, realise version, contact, email etc.
* The API currently supports two types of selectors: equality-based and set-based. A label selector can be made of multiple requirements which are comma-separated.
* Commands:-
* kubectl get pods --selector apps=my-app (to filter applications by my-app) or use -l
* kubectl get pods -l environment=production,tier=frontend {equality-based}
* kubectl get pods -l 'environment in (production),tier in (frontend)' {set-based}

#Taints and Tolerations:

* What pods can be scheduled on a Node. Taint and Toleration are for a node to accept only certain kind of pods.
* If we have a dedicated resource on a node for a particular use case, we like to schedule only those pods that are related to that use-case.
* So we prevent other pods which are not similar from entering the Node by placing a 'Taint'. By placing a taint no pod can schedule on nodes for now.
* Toleration is those pods which can 'tolerate' (or placed on that) node's Taint.
* By default, there is no toleration given to any pod so no pod can be placed on that tainted node.
* We give toleration to a particular pod and give taint to a Node.So only these pods can schedule.
* Remember they can schedule on tainted a node but not necessarily schedule, they may schedule on other nodes too.
* Therefore add a taint in a node and a toleration to a pod.
* taintEffect is what happens to a pod if there is no toleration in them.

taintEffect→ NoSchedule, PreferNoSchedule, NoExecute

* Noschedule:- No pods can be scheduled on this node except for the tolerant pods.
* NoExecute:- No pods can execute on this node except for the tolerant pods.

(comes handy when pods are already running and you tainted nodes with this so tht pods without tolerance to this node can be evicted.)

* PreferNoSchedule: try to not schedule on this node but it is not guaranteed.
* To taint the node use command:-
* kubectl taint node node01 <key>=<value>:taintEffect
* To add toleration in pod (in $.spec):

   tolerations:

- key: "key"

operator: "Equal"

value: "value"

effect: "NoSchedule"

* All values need to be encoded in double quotes"". There is another operator field ‘Exist’ operator which doesn’t require a value, it simply checks if the key exists in the tolerance of the pod.

tolerations:

- key: "key"

operator: "Exists"

effect: "NoSchedule"

* Taints & Toleration just allows certain pods to enter in a node while restricting others but a Pods with toleration doesn't always guarantee to be scheduled on a tainted Node.
* Scheduler never schedules on Master Node as when Kubernetes starts it taints itself with NoExecution taint.
* Commands:-
* kubectl taint node node01 key=value- (remove the taint set on node)
* kubectl describe node node01 | grep -i taint (see the existing taint)

# Node-Selectors:

* In order to run pods to prefer or run in a particular node we use Node select. nodeSelector is the simplest recommended form of node selection constraint.
* We must 1st label the nodes so that in nodeSelector property in Spec we assign those node labels.

-YAML format:-

spec:

nodeSelector:

<key>: <value>

* Commands:-
* kubectl label node <node-name> <key>:<value>
* NodeSeletor is simple and lacks advance features as here we can only assign to one none not set of node and also cannot set any condition.

# Node-Affinity:

* Does the same thing as NodeSelector but with more advanced features and conditions.
* There is also an Anti-Affinity in which cases run against the pods.
* Both node affinity and anti-affinity come under the affinity option under spec.
* There are currently two types of node affinity, called
* requiredDuringSchedulingIgnoredDuringExecution.
* preferredDuringSchedulingIgnoredDuringExecution
* If the labels of node changes or labels are deleted, these types tell api what to do. There is future planning of 3rd type requiredDuringSchedulingRequiredDuringExecution. This type will evict pods from nodes that cease to satisfy the pods node affinity requirements.

-YAML format:-

apiVersion: v1

kind: Pod

metadata:

name: pod-with-node-affinity

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: kubernetes.io/e2e-az-name

operator: In

values:

- e2e-az1

- e2e-az2

preferredDuringSchedulingIgnoredDuringExecution:

preference:

matchExpressions:

- key: another-node-label-key

operator: In

values:

- another-node-label-value

containers:

- name: with-node-affinity

image: k8s.gcr.io/pause:2.0

Here, This node affinity rule says the pod can only be placed on a node with a label whose key is kubernetes.io/e2e-az-name and whose value is either e2e-az1 or e2e-az2. In addition, among nodes that meet that criteria, nodes with a label whose key is another-node-label-key and whose value is another-node-label-value should be preferred.

* You can see the operator In being used in the example. The new node affinity syntax supports the following operators: In, NotIn, Exists, DoesNotExist, Gt, Lt.
* If you specify both *nodeSelector* and *nodeAffinity*, both must be satisfied for the pod to be scheduled onto a candidate node.
* If you specify multiple *nodeSelectorTerms* associated with *nodeAffinity* types, then the pod can be scheduled onto a node if one of the *nodeSelectorTerms* can be satisfied.
* If you specify multiple *matchExpressions* associated with *nodeSelectorTerms*, then the pod can be scheduled onto a node only if all *matchExpressions* are satisfied.
* If you remove or change the label of the node where the pod is scheduled, the pod won’t be removed. In other words, the affinity selection works only at the time of scheduling the pod.

# Resource Requirements & Limits:

* Schedulers place different pods into nodes according to their resource requirements.
* You can specify requirements like cpu, memory, storage etc in the pod definition YAML file. It is called as “Resource Request”
* Since docker containers place no limits on resource usage you specify limits in the definition file and by default, Kubernetes places a limit of 1 CPU to every pod.it is known as “Resource Limit”.
* If you set a memory limit of 4GiB for that Container, the kubelet (and container runtime) enforce the limit.
* Limits and requests for CPU resources are measured in cpu units. One cpu, in Kubernetes, is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors.
* Fractional requests are allowed. A Container with spec.containers[].resources.requests.cpu of 0.5 is guaranteed half as much CPU as one that asks for 1 CPU.
* The expression 0.1 is equivalent to the expression 100m, which can be read as “one hundred millicpu”.
* Limits and requests for memory are measured in bytes. You can express memory as a plain integer or as a fixed-point integer using one of these suffixes: E, P, T, G, M, K.

e.g. 128974848, 129e6, 129M, 123Mi

-YAML format:-

spec:

containers:

- name: db

image: mysql

env:

- name: MYSQL\_ROOT\_PASSWORD

value: "password"

resources:

requests:

memory: "64Mi"

cpu: "250m"

limits:

memory: "128Mi"

cpu: "500m"

-If a pod uses more cpu the it requirement then kubernetes throttle cpu but in case of memory it terminates pod if it continues to exceed memory limits.

# Daemon-Sets:

* DS are like replica sets where you can run multiple instances of pods but it runs one copy of pods to each node.
* It ensures that one copy is present on each node . e.g.
  + a monitoring app or a logging app you deploy a daemonset
  + kube-proxy is deployed as a daemon set.
  + in networking an agent is required in all the nodes.
* In version 1 of kubernetes it uses node-name property it deploys the copy of pods in all nodes as this is a manual scheduling.
* In version v1.12 it uses default scheduler and node affinity properties to deploy copy of pods.

# Static-pods:

* A kubelet can manage the node independently.
* /etc/kubernetes/manifests is a location where you put a YAML definition of a pod so that the kubelet can run it.
* This way of running a pod without an api-server or any other component is called a static pod. you can create a static pod in this way, not a replica set or deployment.
* Kubelet manages these pods on its own like whatever you change it updates it, if you move the file it deletes it etc.
* You can configure in its service file of Linux in /etc/systemd/system/kubeletservice.d & look for parameter:-

"pod-manifest-path=/etc/kubernetes/manifests"

or

It can be done this way using the config option and provide a path to the yaml config file. :-

 --config=/var/lib/..kubeconfig.yaml

 in YAML file--> staticPodPath: /etc/kubernetes/manifests

* If you create a static pod and the kubelet is part of the cluster then the kube-api server still views the details but is read only as you cannot edit it from the master node you have to go to node and edit the YAML file in order to do this.
  + use case: since static pods are not dependent on kubernetes control plane we can deploy kubernetes control planes itself on various nodes by:
  + deploy kubelet 1st from github.
  + from kubelet deploy the pods that use control planes images by creating config files as apiserver.YAML, etcd.YAML etc.
* To delete it ssh into the node and look for the config file and look at the static path parameter and then go to that path and then delete the YAML file.

# Multiple-scheduler:

* You can create your own scheduler to place pods on nodes according to your conditions.
* If you running as standalone kube-schedule.service then you can modify the parameter of service:-

  --scheduler-name= my-custom-scheduler or default-sheduler

or

If you are using kubeadm as it deploys controlplane component as static pods so in staticPodPath=/etc/kubernetes/manifest directory change the same property that is

-scheduler-name= my-custom-scheduler or default-sheduler in $.spec section of my-custom-scheduler.yaml!

* Put schedulerName: my-custom-scheduler property in pods so that this scheduler schedules this pod like:-

spec:

schedulerName: my-custom-scheduler

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

* Commands:-
* kubectl get event (to get all event from current namespace)
* kubectl logs my-customer-scheduler (to get logs of your scheduler)

Note: --leader-elect=true is for multiple Master in HA env where you select a leader which schedules the pods, set it to false and if you have multiple Master then set it to true & provide --lock-object-name=my-custom-scheduler so to differentiate between this scheduler on letting a leader in multiple master node in HA env.

# Configure a scheduler:

* Manually by using a binary, execute it in linux env just like any other executable.
* Using a kubeadm deploy a static pod and configure properties (ready with docker image build used by the pod which contains your binary.).

## Logging & Monitoring:

# Monitor Cluster Component:

* Monitor memory, cpu, storage, temp etc utilization of a Node as well as pods and their performance.
* There is no monitoring solution provided by Kubernetes.
* Kubelet contains a sub-component called "cAdvisor '. This is responsible for fetching all metrics of pods and giving them to the kube-api server. there by any 3 party solution picks up those metrics eg Metrics server.
* Commands:-
* kubectl top node
* kubectl top pod
* kubectl addons enable metrics- server (for minikube)

# Logging :

* Event simulator is a docker container which generates event logs.
* Commands:-
* docker logs -f event-simulator to see logs.
* kubectl logs -f name-of-pod -c container-name

## Lifecycle Management:

# Rolling Update and Rollback:

* Step by step app down and then the new upgraded version is up is called Rolling update.
* Whenever we make changes to the image or any YAML file parameter a rollout is triggered and a revision is saved.
* When a new rollout of deployment is triggered, kubernetes creates a new replica set containing upgraded apps & changes while simultaneously bringing down the app in original replica set.
* Recreate strategy is all app instances are down and new are all up together this creates app downtime, by default in Kubernetes rollout strategy is used.

Commands:-

* kubectl apply -f deployment-name.yaml (declarative change)
* kubectl set image deployment/my-deployment <container-name>=<image:version> (imperative change, only affects live object not the yaml file)
* kubectl rollout status deployment/my-deployment
* kubectl rollout history my-deployment
* kubectl rollout undo deployment/my-deployment

# Configuring Application:

* Configuring applications comprises of understanding the following concepts:
* Configuring Command and Arguments on applications
* Configuring Environment Variables
* Configuring Secret
* Like there are Commands, arguments, and Entrypoints in Docker there is a corresponding property in Kubernetes which will override these.
* Note in docker ENTRYPOINT is a command/script that is run whenever docker container starts running and whatever arguments you pass it will append to ENTRYPOINT in Docker but If you don’t specify any argument then default argument in CMD is used.
* eg: *docker run ubuntu 10* (Here we added sleep command to Entrypoint) no need to explicitly mention sleep command just pass the argument.

If we run like this *docker run ubunt*u then without argument in Entrypoint there's error so you passed argument in *CMD["5"]* this gets appended when there is no argument given as default argument.

* Anything that is appended to the Docker run command will go to the "args" property in the form of an array in spec and to append a command we use the "command" property in spec in the YAML definition file.
* This args property will override the CMD default argument and the command will replace the Entrypoint in Docker.

-YAML format:-

spec:

containers:

- name: ubuntu-sleeper

image: Ubuntu:16.1.0

command: ["sleep"]

args: ["10"]

# Environment variables:

* In docker we use: docker run -e app\_color=blue web-app to set env variable.
* Tdo set an environment, add an env property in spec, env is an array.

-YAML format:-

spec:

containers:

- name: envar-demo-container

image: gcr.io/google-samples/node-hello:1.0

env:

- name: DEMO\_GREETING

value: "Hello from the environment"

- name: DEMO\_

FAREWELL

value: "Such a sweet sorrow"

* It is a plain key-value format, there are others ways you can set env variable

like configMaps or secrets

# Configuration Maps:

* When you have a lot of pod definition files it hard to manage all env variables stored in various pod files.
* It is used to centralize the configuration data in key-value pairs.
* A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.
* A ConfigMap can allow you to decouple environment-specific configuration from your container images, so that your applications are easily portable.
* Commands:-
* kubectl create configMap app-config --from-literal=app\_color=blue --from-literal=app\_type=front-end
* kubectl create configMap app-config --from-file=<file-path> (specify file that contains required data)
* Use these commands or create a configMap definition file (declarative way), this file doesn't contain spec instead has data section:-
* There are four different ways that you can use a ConfigMap to configure a container inside a Pod:-
  + Command line arguments to the entrypoint of a container
  + 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

-YAMLformat

apiVersion: v1

kind: ConfigMap

metadata:

name: game-demo

data:

player\_initial\_lives: 3

player\_max\_lives: 25

-in pod specify config map as:-

   spec:

containers:

- name: demo

image: game.example/demo-game

env:

- name: PLAYER\_INITIAL\_LIVES

valueFrom:

configMapKeyRef:

name: game-demo *# The ConfigMap this value comes from*

key: player\_initial\_lives *# The key to fetch*

*or as all env taken as :-*

envFrom:

configMapRef:

name: game-demo

*or as volume type to mount configMap file :-*

spec:

containers:

- name: demo

image: game.example/demo-game

volumeMounts:

- name: config

mountPath: "/config"

readOnly: **true**

volumes:

*# You set volumes at the Pod level, then mount them into containers inside that Pod*

- name: config

configMap:

*# Provide the name of the ConfigMap you want to mount.*

name: game-demo

# Secrets:

* Secrets are used to store sensitive information like password of database etc.
* Secrets are similar to configmap except they are encrypted , hashed or encoded.
* In Secrets values are encoded in Base64 format.
* Commands:-
* kubectl create secret generic app-secret --from-literal=db\_passsword=mysql123 --from-literal=db\_host=mysql
* kubectl create secret generic app-secret --from-file=<file-path> (specify file that contains required data)
* echo -n ‘<value>’ | base64 (paste the output in the corresponding key)
* echo -n ‘<value>’ | base64 --decode to decode it.

-YAML format:-

# data:

username: user1

password: user123 this data must be in encoded format #

apiVersion: v1

kind: Secret

metadata:

name: mysecret

data:

username: YWRtaW4=

password: MWYyZDFlMmU2N2Rm

* In pod definition file secrets links the same as configMaps.

# Multi-container pods:

* -A times you need more than one container in a pod like logging app and a web app

they share the same lifecycle, network space, namespace etc.

* They are created together and destroyed together, refer to each other as local host and share same storage space.
* That is why $.spec.containers[] is an array to specify multiple containers.

# Init container:

* At times you may want to run a process that runs to completion in a container after the container starts.
* Init containers 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.

-YAML format:-

* 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;']

# Self Healing Applications:

* Kubernetes supports self-healing applications through ReplicaSets and Replication Controllers.
* Kubernetes provides additional support to check the health of applications running within PODs and take necessary actions through Liveness and Readiness Probes.

## Cluster Maintenance:

# OS Upgrades:

* Takedown a node for upgradation of OS running in Node.
* If a node fails then the pod serving applications also down and user might be impacted so in this case if nodes comes up immediately then kubelet restarts again running all pods.If node does not come up then kubernetes considers them dead, if pods are part of replica set then kubernetes starts them in other nodes.
* Pod eviction time is 5 mins i.e. it waits for 5 mins for a node to come online. You can do a maintenance task if you are sure that it is completed in 5 mins, If not you can drain the nodes so that pods are evicted out of the node and run on another.
* Commands:-
* Kubectl drain node-1
* Kubectl uncordon node-1 (to reschedule node after it comes back online.)
* Kubectl cordon node-1 (it just marks the node unschedulable.No new pods now can schedule on this node and the existing one keeps running.)
* Here the pods that are moved to another node are not moved again to this node.
* Also you cannot delete Pods not managed by ReplicationController, ReplicaSet, Job, DaemonSet or StatefulSet if running on node.
* If a lone pod is running and node is drained with --force flag than the pod is gone forever.

# kubernetes Software versions:

* -v1.18.2 , here v1 is major version, 18 is minor version & 2 is patch version
* Minor versions are released every few months with new functionalities.
* Patches are released more often with critical bug fixes.
* The ETCD cluster and coreDNS are different projects from kubernetes and have their own versions.

# Cluster Upgrade Process:

* kube-api server is where all components depend so none of components should be higher than kube-api server.
* kube-api: X version, kube-controller→ X to X-1, kube-scheduler→ X to X-1,kubelet & kubeproxy→ X to X-2 and kubectl → X+1 to X-1.
* Kubernetes supports upto recent 3 minor versions, so when you are 3 minor versions behind it is a good time to upgrade.
* Upgrade depends on how your kubernetes cluster is set up e.g. google cloud upgrade by just a few clicks, kubeadm has a kubectl upgrade plan and apply or manually from scratch you have to update all your cluster components.
* Upgrading kubernetes cluster has 2 steps:

1. Upgrading Master Node: while upgrading master node, all control plane components go down briefly but here the pod running in worker node and worker node itself has no impact.
2. Upgrading Worker Nodes: there are 3 strategies to upgrade worker nodes.

* Upgrading all the worker nodes at once. This causes application downtime.
* Upgrade one by one : this while moving your pods to other nodes so this will not impact your applications.
* Lastly this strategy requires to deploy a new node and move workload to a new node after installing new versions and decommissioning the old nodes. This strategy is very helpful in cloud environments in which you can easily provision a new node.
* Upgrade the Master Node steps:-

1. Upgrade kubeadm tool itself → ***apt-get upgrade -y kubeadm=1.12.0-00***
2. Upgrade kubernetes Control-Plane Components → ***kubectl upgrade apply v1.12.0 --ignore-preflight-errors=ControlPlaneNodesReady***
3. Upgrade kubelet (running in Master Node): if you install kubeadm then a kubelet is
4. installed in your master node → ***apt-get -y upgrade kubelet=1.12.0-00***
5. Restart kubelet service→ ***systemctl restart kubelet***

-Note: here if you fire kubectl get nodes the current version still shows the old version If you have not upgraded the kubelet on Master which is running as a service by kubeadm tool in Master Node.

* Upgrade the Worker Nodes steps:-

1. Drain the node → ***kubectl drain node01*** (This will terminate all pods and cordon the node and reschedule them on other nodes.)
2. Upgrade kubeadm & kubelet on worker node→***apt-get upgrade -y kubeadm-1.12.0-00***

***apt-get upgrade -y kubelet=1.12.0-00***

***(don’t use -y if downgraded or authenticate error or any error occurs . if want to use -y → downgrading use --allow-downgrades with -y and if authentication error use --allow-unauthenticated with -y)***

1. Upgrade node configuration→***kubeadm upgrade node config --kubelet-version v1.12.0***
2. Restart kubelet service→ ***systemctl restart kubelet***
3. Mark node schedulable: when you drained the node it marked as unschedulable, make it schedulable by→ ***kubectl uncordon node01***

# Backup & Restore process:

* To backup: resource configurations,ETCD cluster,persistent volumes.
* Always have a declarative approach to create pods so that you can save definition files and backup, usually backup in a git repository.
* If your created by a imperative approach then use → ***kubectl get all --all-namespace -o YAML > all-objects.yaml***
* All Kubernetes objects are stored on etcd. Periodically backing up the etcd cluster data is important to recover Kubernetes clusters under disaster scenarios, such as losing all master nodes. The snapshot file contains all the Kubernetes states and critical information.
* Instead of saving every resource object you can choose to save the entire ETCD cluster,ETCD cluster is hosted on master node, data is stored through parameter in ***etcd.service.d --data-dir=/var/lib/etcd***. If kubeadm it is in etcd static pod.
* You can take a snapshot of etcd by→ ***etcdctl snapshot save etcd\_save.db (use with etcdctl version 3)*** and whenever a server crashes use the most recent snapshot to restore the cluster.

This command is used with lot of important flags :-

* ***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 /backup/etcd-snapshot-1.db***

*Since our ETCD database is TLS-Enabled, the following options are mandatory:*

*--cacert→ verify certificates of TLS-enabled secure servers using this CA bundle*

*--cert → identify secure client using this TLS certificate file*

*--endpoints=[127.0.0.1:2379] → This is the default as ETCD is running on master node and exposed on localhost 2379.*

*--key→ identify secure client using this TLS key file*

* To restore etcd cluster:-

1. Stop the kube-api server service→ Service kube-apiserver stop

If the kubeadm tool removes the static pod yaml file.

*// etcdctl is a command line client for etcd.*

*To make use of etcdctl for tasks such as backup and restore, make sure that you set the ETCDCTL\_API to 3.*

*export ETCDCTL\_API=3 , etcdctl snapshot save -h and keep a note of the mandatory global options.*

*//*

1. Restore snapshot that was saved earlier→ etcd snapshot restore

Restore also contains these flag we used earlier and additional flags:-

* ***ETCDCTL\_API=3 etcdctl --endpoints=https://[127.0.0.1]:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt \***

***--name=master \***

***--cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key \***

***--data-dir /var/lib/etcd-from-backup \ (this directory you specify in etcd.yaml after restoration)***

***--initial-cluster=master=https://127.0.0.1:2380 \ (Get from etcd.yaml pass both localhost and ip of master)***

***--initial-cluster-token etcd-cluster-1 \ (this parameter you specify in etcd.yaml after restoration, save it)***

***--initial-advertise-peer-urls=https://127.0.0.1:2380 \ (Get from etcd.yaml pass both localhost and ip of master)***

***snapshot restore /tmp/snapshot-pre-boot.db***

1. Here when etcd is restored from snapshot it initializes new cluster configuration and configures members of etcd as new members of this new cluster.this so that new member won’t join the current system.

e.g. → if you restore this snapshot for testing purposes only then you don’t want the testing members to join the existing production server.

1. In order to ensure this we use flag ***--intial-cluster-token***  and specify a new token. You have to configure this newly created ETCD clusters from restoration in etcd.service or etcd static pod in kubeadm. → provide ***--initial-cluster-token=etcd-1 ,***
2. Reload the service daemon and restart the etcd service→ ***systemctl daemon reload & systemctl restart etcd.service, data-dir=/var/lib/etcd-from-backup***

For Kubeadm make changes to restore the new etcd backup at /etc/var/lib/etc-from-backup (or wherever you put data using --data-dir option), update changes in etcd.yaml file at /etc/kubernetes/manifests/etcd.yamletcd is a docker container use docker ps to check if it is restarted.

1. Finally start the kube-api service → ***systemctl start kube-apiserver or service kube-apiserver start***

For kubeadm paste the static pod that you moved again here it will be automatically started.

## Security:

# Kubernetes security primitives:

* All Host (servers) must be secure like root access disabled, etc
* there 3 types of measures in security:
* who can access the data: Authentication:-
  + By static files of user and password
  + By users and tokes
  + By Certificates
  + Service accounts
  + External Authentication service: LDAP
* what can they do: Authorization:-
  + RBAC→ Role Based Access Control
  + ABAC→ Attribute ---”-----”----”----
  + Node authorization etc
* All communication between the various components of the cluster are secured by TLS certification authentication and encryption.
* Networks Policies are there to restrict communication between pods across clusters.

# Authentication:

* Kubernetes cluster is directly accessed by admin or developers as the end user security is in application.
* Bots are authenticated using service accounts.
* Kubernetes has no way of creating users in a cluster to authenticate.
* Static password or token file based authentication can be done by passing parameter --basic-auth-file=user.csv or --token-auth-file=token.csv, once you created user password or token file.

# TLS Certificate:

* Certificate is used to guarantee trust between two parties during a transaction.eg during web access communication between end user and server is encrypted and to authenticate whether the server is what the server says.
* Basics: symmetric encryption (1 key to encrypt data and is sent to the server) is used for communication and Asymmetric encryption {i.e. https} is user firstly secure symmetric encryption (2 keys private and public key is used. Public key of the server firstly goes to the end user to encrypt the single symmetric key and then this encrypted symmetric key is decrypted by the server's private key. Now both parties can communicate with an encrypted network.)
* But using the above method if the hacker posed as an actual server then ? how do we know that the server end user communicating with server is actually the real server not some fake one?
* That's when a certificate comes into play. A certificate says that this is me and here a certificate to prove it. Certificates contain various fields like name, domain, etc but anyone can make them then what??
* That’s when the most important part of certification comes in and that is “issued by” field. A self signed certificate is you are the one who signed his own certificate which is not secure (remember when chrome warns about connection is not private i.e. the server you are connecting to gives you a self signed certificate.). Certificates are signed by a well-known authority called Certificate Authority (abbrv. as CA) and there are few of them in the world.
* Now comes the question: how does the browser verify that the certificate signed by CA is legit? → Here when signing certificates, CA signed them by using their private key and their public key are stored in the browser themselves.
* Here the server is validated as they are the actual server user is trying to connect to ? how the server knows that the user is who they are ??
* Here also the same way a client certificate is generated signed by CA under the hood and how infrastructure of managing certification, encryption etc is known as “Public Key Infrastructure” (PKI).
* Naming conventions:- Certification with public key→ .crt or .pem extension eg server.pem , server.crt etc.

Certificate with private key--> .key or -key.pem extension eg server.key or server-key.pem

# TLS certificate in kubernetes:-

* 3 types of certificates:
* server certificate:-configured on server
* client certificate:-configured on client
* root certificate:-configured on root.(certificate signed by CA)
* So In TLS security every component (client/server/bot/user/admin etc.) needs a pair of certificate (signed by CA) and a private key and also a CA certificate public copy in order for them to validate each other.
* In kubernetes in order to communicate each must have a TLS security authentication i.e. each server must talk to an api-server with certificates and keys and the api server has its own set of certificates and a private key.
* For api server administrator,users (ourselves ) who try to talk to api and give commands need to authenticate with certificate and private key adm.cert and adm.key. kube -api server treats each component as client so scheduler has scheduler.cert and scheduler.key as all components have their own set of certificate and key in order to communicate.
* Generate certificate by:-EasyRSA, OpenSSL etc

Eg with OpenSSL:- For root level 1st ie for CA:-

* generate private key→ *openssl genrsa out ca.key 2048*
* certificate signing request→ *openssl req -new -key ca.key -subj “/CN=KUBERNETES-CA” -out ca.csr*
* generate certificate→ *openssl x509 -req -in ca.csr signkey ca.key -out ca.cert*
* Now we will use this ca.key private key and ca.cert self-signed certificate in all other certificate to sign them, here are the steps for client Admin user:-
* generate private key→ *openssl genrsa out admin.key 2048*
* certificate signing request (here system-master is a group assigned in a certificate)→ *openssl req -new -key admin.key -subj “/CN=KUBE-ADMIN****/OSYSTEM:MASTERS****” -out admin.csr*
* generate a certificate which is signed by CA (created by us.) → *openssl x509 -req -in admin.csr -CA ca.cert -CAkey ca.key -out ca.cert*
* In order to validate request with api admin can use a api call by curl command as→ *Curl* [*https://kube-apiserver:6443.api/v1/pods*](about:blank) *--key admin.key --cert admin.cert --cacert ca.cert.*
* Or configure a YAML file called ‘kube-config.yaml’ and specify endpoints details and certificates and keys.
* Here all components also in order to verify each other need a copy CA certificate as well to validate.
* Server certificate:-e.g. ETCD server same process of generating and signing certificate.

With the HA cluster we need additional peer certificates.

* For an api-server the same process applies, it also needs both client and server certificates to act as both.
* View certificates:

To view certificates depends on how kubernetes is deployed:-

* The hard way:-cat /etc/systemd/system/kube-apiserever.service
* kubeadm:- cat /etc/kubernetes/manifests/kube-apiserver.yaml (static pod path)
* Look for path of all certificate and view them.

# Certificates API:

* If a new user wants to access the cluster as suppose the admin has only access so the new user creates her own private key and certificate and in order to sign it she applied sign requests to admin and the admin takes sign request to ca server to sign by its certificate and ca’s private key.
* So CA server is basically a pair of certificates and keys, so anybody who has them can access the kubernetes cluster so these files need to be in a safe place in a high security server known as a CA server.
* Since a certificate placed on Master Node is the CA server itself, kubeadm tool also places CA certificates on master node.
* As the user grows you can’t do lots of signing certificates manually so kubernetes has a certificate api which manages certification tasks like CSR (certification Signing Request).
* CSR for new user steps:

1. User first creates its own private key by→ *openssl genrsa -out newuser.key 2048*
2. User then generate its own certificate & sends signing request to Admin→ *openssl req -new -key -subj “CN=newuser” -out newuser.csr*
3. Admin then take the key and create a Certificate Signing Object (*it is like any other object in kubernetes with kind=CertificateSigningRequest.)*
4. ***Kubectl create -f newuser-csr.yaml*** then***kubectl get csr*** to see all signing requests then***kubectl certificate approve new-user*** *.*

* In controler manager there 2 component that do these tasks:-
* CSR-Approving controller
* CSR-Signing controller
* If anyone has to sign certificates then they need root CA ‘s certificate and key, in kube-controller-manager.yaml you can set these 2 properties and locate the path where is CA certificate and key are placed:-
* --cluster-signing-cert=/etc/kubernetes/pki/ca/ca.cert
* --cluster-signing-key=/etc/kubernetes/pki/ca/ca.key
* Commands:-
* kubectl create -f user-csr.yaml
* kubectl get csr
* kubectl approve user

# KubeConfig:

* In order to hit api-server for certificate config, csr config, etc we passed them as an option to curl for user authentication.
* whenever you fire a command like kubectl get pods → pass the adm.cert, adm.key & ca.cert to authenticate and query the api server by passing this parameters everytime is not viable so kubernetes has a config file called ’kubeconfig.yaml’ which he looks every time a user query so that it authenticates the request.
* If you have this file at default location you don’t need to pass --kubeconfig= option in query but if the file is somewhere else then specify → kubectl get pods --kubeconfig /<path-to-file>/config.yaml
* The default file is located in /$HOME/.kube/config which is picked up by kube-controller thus if the file is placed in another folder then you have to explicitly mention path also.
* kubeconfig file has 3 parts:
* Clusters:-these are kubernetes clusters that are being used by a user.
* User:- users which are going to access the clusters, this contains user certificates and keys etc.
* Context:-this part joined the above 2 parts eg admin uses a production server Admin@prod is a context.

-Kubeconfig YAML format:-

apiVersion:v1

Kind: Config

currentcontext: admin@my-kube-playground # name of default context to use

clusters:

* name: my-kube-playground

cluster:

certificate-authority: /etc/kubernetes/pki/ca.cert

server: https://my-kube-playgound:6443

context: # here @ is a convection for ease, you can name it anything

* name: admin@my-kube-playground

context:

cluster: my-kube-playground

user: admin

namespace: my-app #you can specify a namespace

users:

* name: admin

user:

client-certificate: /etc/kubernetes/pki/users/admin.cert

client-key: /etc/kubernetes/pki//users/admin.key

# In clusters.name[].certificate-authority you can pass certificate-authority-data and paste the full certificate right here but encode it in Base64 first.

* commands:-
* kubectl config view <> (command to view config)
* kubectl config use-context dev@my-kube-playground (command to change the current context, This will reflect in YAML file)
* kubectl config -h (other options)

## Authorization:

# API groups:

* there 2 main things in api-version components /api & /version.
* /api si sub-divided into many parts such as /logs, /version, /health, /metrics, /apis etc
* /apis & /api are responsible for cluster functionality./api is a core api group and contains /v1 which contains namespaces, events, pods, secrets, service etc & /apis is named api group.
* /apis contains newer features and is more organized. It contains /apps→ /v1→ /deployments, /replicasets, /statefulsets; /extensions; /networking.k8s.io→ /v1→ networkpolicies; /storage.k8.io; /authentication.k8s.io etc.
* With these objects such as /deployment it has a set of associated actions such as ‘delete’, ‘list ', ‘update’, ‘get’, ‘create’ etc these are called “verbs”.
* in order to access thee we need authentication by 3 methods:
  + curl <https://localhost:6443> -k --key admin.kye --cert admin.cert --cacert ca.cert
  + use kube proxy service to access this by→ kubectl proxy , this uses server certificates to access api.thus kubectl proxy uses por 8001 so curl [https://localhost:](https://localhost:6443)8001 -k will be sufficient. here kubectl proxy (is proxy service) ==! kube-proxy component.
* So conclusion all Resources in kubernetes are grouped into different api groups and at the top level you have core /api group and named /apis group

# Role Based Access Controls (RBAC):

* You can create a role for a user so that it can access a few items.
* A Role always sets permissions within a particular namespace; when you create a Role, you have to specify the namespace it belongs in.

-YAML format:-

**apiVersion**: rbac.authorization.k8s.io/v1

**kind**: Role

**metadata**:

**namespace**: default

**name**: pod-reader

**rules**:

- **apiGroups**: [""] *# "" indicates the core API group*

**resources**: ["pods"]

**verbs**: ["get", "watch", "list"]

**resourceName**: [“pod1”,”pod2”] #specify particular name of resource

-Next step is link the user to the role and for this you create another object known as **Role Binding**.

**apiVersion**: rbac.authorization.k8s.io/v1

*# This role binding allows "jane" to read pods in the "default" namespace.*

*(can only ready pod1 and pod2 as resource Name is specified)*

*# You need to already have a Role named "pod-reader" in that namespace.*

**kind**: RoleBinding

**metadata**:

**name**: read-pods

**namespace**: default

**subjects**:

*# You can specify more than one "subject" , provide user details here*

- **kind**: User

**name**: jane *# "name" is case sensitive*

**apiGroup**: rbac.authorization.k8s.io

**roleRef**: # provide the role the user is bound to!

*# "roleRef" specifies the binding to a Role / ClusterRole*

**kind**: Role *#this must be Role or ClusterRole*

**name**: pod-reader *# this must match the name of the Role or ClusterRole you wish to bind to*

**apiGroup**: rbac.authorization.k8s.io

* Both are limited to namespaces (meaning they are created in a namespace) so for a particular namespace define namespace property in metadata.
* Commands:-
* kubectl get roles
* kubectl get rolebindings
* kubectl describe role role-name (more details about roles)
* kubectl describe rolebinding devuser-developer-binding
* kubectl auth can-i create deployments # verb resource-name (for your access to view)
* kubectl auth can-i create deployments --as admin (to see access as the same user) --namespace prod (also specify namespace).
* kubectl create role pod-reader --verb=get --verb=list --verb=watch --resource=pods (create a Role named "pod-reader" that allows user to perform "get", "watch" and "list" on pods)
* kubectl create role pod-reader --verb=get --resource=pods --resource-name=readablepod --resource-name=anotherpod (with ResourceName specified)
* kubectl create role foo --verb=get,list,watch --resource=rs.extensions (with API Group specified)
* kubectl create role foo --verb=get,list,watch --resource=pods,pods/status (with SubResource specified)
* kubectl create rolebinding dev-user-binding --user=dev-user --role=developer (specify user name and the role to bind to with rolebinding imperatively)

# Cluster roles and bindings:

* Nodes cannot be associated with a namespace. They are cluster scoped, other things that are cluster scope are certificate signing requests, PV, services etc. to get namespaced resources only command:-
* kubectl api-resources --namespaced=true
* How do we authorize cluster scoped resources?

we use **ClusterRole** to provide access to cluster scoped resources eg cluster-admin role to delete,drain,view Nodes or a storage-admin to view,delete,create Persistent Volumes.

-YAML format:-

**apiVersion**: rbac.authorization.k8s.io/v1

**kind**: ClusterRole

**metadata**:

*# "namespace" omitted since ClusterRoles are not namespaced*

**name**: secret-reader

**rules**:

- **apiGroups**: [""]

*#*

*# at the HTTP level, the name of the resource for accessing Secret*

*# objects is "secrets"*

**resources**: ["secrets"]

**verbs**: ["get", "watch", "list"]

* ClusterBinding:-Bind the cluster role created. Both are the same as role-binding except kind property.

-YAML format:-

**apiVersion**: rbac.authorization.k8s.io/v1

*# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.*

**kind**: ClusterRoleBinding

**metadata**:

**name**: read-secrets-global

**subjects**:

- **kind**: Group

**name**: manager *# Name is case sensitive*

**apiGroup**: rbac.authorization.k8s.io

**roleRef**:

**kind**: ClusterRole

**name**: secret-reader

**apiGroup**: rbac.authorization.k8s.io

* Here it is not necessary that you define only a cluster scoped role in the definition file you can set namespaced resources too but now that user can have access across all namespaces.

# Image Security:

* Image: docker.io/nginx/nginx here 1st part→ registryname, 2nd part→ user account name, 3rd part→ imagename, google has its own registry as gcr.io for pulling images.
* Using a private registry is a good security measure and many cloud providers give it.
* to use a private registry from docker’s perspective you first login to your registry as:-
  + *docker login my-private-registry.io*
  + *docker run my-private-registry.io/apps/internal-app*
* How do you implement the authentication to your private registry?:-
* *create a secret object :- kubectl create secret docker-registry regcred --docker-server= my-private-registry.io --docker-username= --docker-password= --docker-email=*
* here this secret is of type docker registry and is built-in to keep your docker credentials.
* you then pass then in pod definition file in spec.imagePullSecrets given below as:-

-YAMl format:-

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: private-reg

**spec**:

**containers**:

- **name**: private-reg-container

**image**: my-private-registry.io/apps/internalapps

*imagePullSecrets:*

- **name**: regcred

* Commands:-
* kubectl create secret docker-registry regcred --docker-server=<your-registry-server> --docker-username=<your-name> --docker-password=<your-password> --docker-email=<your-email>

# Security Context:

* When you run a docker container you have option to define a set of security standards such as id of user that runs the container→ docker run --user=db-admin mysql
* You can also choose to configure security settings at pod level.in spec.securityContext as:-

**spec**:

**securityContext**:

**runAsUser**: 1000

**containers**:

- **name**: mysql

**image**: mysql:6.9

or move this fields at a container level as:-

**spec**:

**containers**:

- **name**: ubuntu

**image**: ubuntu:16.7

**securityContext**:

**capabilities**:#linux capabilities added here only at container level

**add**: ["NET\_ADMIN", "SYS\_TIME"]

# Network Policy:

* There are two types of traffic conserining kubernetes:-
* Ingress:- For a web-server it **receives incoming** traffic from the user is ingress traffic.
* Egress:-For a web-server again it **sends outgoing** requests/traffic to the app-server is Egress traffic.
* Here response doesn't matter the direction of origin is.
* By default all pods are able to communicate with others and if we want some pods to not communicate with a particular pods we need NP.

eg:- Front-end web-server and backend DB we don't want db to communicate directly with front-end web server.

* Network Policy is another object in kubernetes, you link a NP to one or more pods. You can define rules in NP. We link them by labels & selectors.

eg Allow ingress traffic from backend-server-api to DB

-YAML format:-

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: db-policy

namespace: default

spec:

podSelector:

matchLabels:

role: db

policyTypes:

- Ingress

- Egress

ingress:

- from:

- podSelector:

matchLabels:

role: frontend

ports:

- protocol: TCP

port: 6379

egress:

- to:

- ipBlock:

cidr: 10.0.0.0/24

ports:

- protocol: TCP

port: 5978

* Note here NP are enforced by Network solutions are not all of them supports it eg Flannel doesn't support it while Calico, Romana, Weave-net supports it.

## Storage

# Brief Recall of storage with container(Docker):

* /var/lib/docker is where docker keeps all files related to containers info, system info, containers, images etc.
* Since data is deleted in container when it is terminated so we wish to persist our important data so that after container is exit data remains.We can do this in docker by creating a volume, this new folder is saved in /var/lib/docker/volumes/\*volume|storage\_name\*
* Storage drivers help manage storage in image, container etc. but volumes we attach or create are not maintained by storage drivers they are managed by volume drivers.
* eg Loca volume plugin helps create volume on docker host, Azure file storage, Convoy, Rexray etc.

# Container Storage Interface:

* Container Runtime Interface how an orchestration solution like kubernetes would communicate with container solutions like Docker, SO in future new Container solutions developed then they have a standard CRI to follow.
* Similarly for Networking solutions CNI (Container Networking Interface) was introduced as now we have flannel, weaveworks etc.Therefore CSI Container Storage Interface was developed to support multiple storage solutions as portwox, Amazon EBS, GlusterFS etc.
* CSI is meant to be a universal standard not only kubernetes but Mesos etc implemented it.
* It has set of RPCs(Remote Procedure Calls) eg. If a container needs new volume then an RPC is initiated , the storage driver plugin should implement this RPCs.

# Volumes(Storage):

* To persist processed data in the container we attach a Volume so when data is created it gets stored in this volume so same with the pods.

-YAMl format:-

**spec**:

**containers**:

- **image**: k8s.gcr.io/test-webserver

**name**: test-container

**volumeMounts**:

- **mountPath**: /test-ebs

**name**: test-volume

**volumes**:

- **name**: test-volume

*# This AWS EBS volume must already exist.*

**awsElasticBlockStore**:

**volumeID**: <volume-id*>’*

*fsType: ext4*

*OR*

**hostPath:**

**path:** *data*

**type:** *Directory*

# Persistent Volume:

* When you have a large env with a large user and lots of pods, users have to configure storage every time for each pod.instead creates a large pool of storage and users carved out volumes from it.
* A PV is a cluster wide pool of storage volumes configured by an admin to be used by users deploying apps on a cluster. The user can now use this pool using a PVC Persistent Volume Claim.

-YAML format:

**apiVersion**: v1

**kind**: PersistentVolume

**metadata**:

**name**: pv0003

**spec**:

**capacity**:

**storage**: 5Gi

**volumeMode**: Filesystem

**accessModes**:

- ReadWriteOnce

**persistentVolumeReclaimPolicy**: Recycle # by default is Retain

# Persistent Volume Claim:

* PVC is a storage cut in PV used by apps.
* Every PVC is bound to a single PV.
* Kubernetes matches PVC to a best PV if there are multiple PV matches to PVC then you can still use Labels and selector to bind to a particular one.

-YAML format:

**apiVersion**: v1

**kind**: PersistentVolumeClaim

**metadata**:

**name**: myclaim

**spec**:

**accessModes**:

- ReadWriteOnce

**volumeMode**: Filesystem

**resources**:

**requests**:

**storage**: 8Gi

* If we delete a PVC then by default the object is deleted but storage data (PV) remains we can choose to delete by : “persistentVolumeReclaimPolicy: Retain” set it to delete.Here PV if deleted is not available to bind. If we set it to the ‘ Recycle’ option then data is deleted and PV is also available.
* Using PVC in a pod:
* apiVersion: v1
* kind: Pod
* metadata:
* name: mypod
* spec:
* containers:
* - name: myfrontend
* image: nginx
* volumeMounts:
* - mountPath: "/var/www/html"
* name: mypd
* volumes:
* - name: mypd
* persistentVolumeClaim:
* claimName: myclaim

## Networking In Kubernetes:

# Switching & Routing:

* To connect two systems we need a switch and a switch creates a network and to connect to a switch we need interfaces on both systems.
* To see interface (on linux) use command ***ip link***
* We need the ip address of both systems( their interfaces) to be the same(not identical just be in the same network) as the ip address of the switch. Switch then forwards packets to the system in the network.
* So far the system in a particular network is connected in order for the two networks to be connected we need a router.
* The router therefore has more than one ip address of different networks in order to forward packets between networks.
* In order for a system to connect to a system with a different network it needs to know where the router is to send the packed through it. Router is like just another device on the network and in order to find it that’s where Gateway comes into play. so what happens is when a system needs to communicate with system outside the network looks through various routes i.e. gateways and if network ip of a system that it has to send to matches with a gateway ip then it send packet through that gateway (which is one of interface ip of a router) and the router now forwards this packet.
* Gateway is like a door to the outside world, basically how the internet works.The gateway is nothing but an Interface IP of a router i.e. a door to a vehicle and this vehicle transports you to another location.

***route command*** *(to know current routing to different networks)*

***ip route add <Network(cidr)> via <Gateway-IP>*** *(router connect different network and has interface points il all of them so that different hosts in different network can talk to each other through this interface called gateway)*

* So if your router is connected to internet and you need to open google ip=172.217.194.0 add a route by command(your gateway ip-192.168.2.1):

***ip route add 172.217.194.0/24 via 192.168.2.1***

* This Interface Point of router which is connected to the internet which can basically route any network it can get because it is inter-connected to every network (i..e internet) is called the Default Gateway. Everytime time it doesn’t find a network route in the routing table for that network it automatically forwards this to the Default Gateway.

To make gateway a default gateway we use command:

***ip route add default via 192.168.2.1***

Instead of default we can say 0.0.0.0 it means any ip destination.

* As mentioned if they are more then two router say one is for the internet and one is for the internal private network and they are separate entries for each network as:

*you are a 192.168.2.0/24 network*

*two router have address:192.168.2.1 and 192.168..2.2 (you can also have 2 different interfaces of computer which contains different IP’s if your router address are completely different so one of them falls into your private network you can build and another one for internet to connect to all other networks)*

*ip route add default via 192.168.2.1 → add default to connect to all other networks (internet)*

*ip route add 192.168.1.0/24 via 192.168.2.2 → add this gateway to connect to 192.168.1.0/24 Network.(private network , you are in 192.168.2.0/24 network)*

Note: ***-ip route add this\_network via my\_gateway\_or\_router\_interface\_point***

-gateway ip always falls into your network range.

* In linux suppose we have 3 systems connected system A has ip=192.168.1.5 and C has a ip=192.168.2.5 and a system B which has 2 interfaces adapter has 2 ips adrs=192.168.1.6 &192.168.2.6 so A can send msg and receive them to C via B. But in Linux this Ip forwarding is by default not allowed; you have to enable it for security reasons.

# DNS:

* In /etc/host associate an IP address to a name any name will do in order to communicate using that name instead of pinging the IP. Can have multiple hosts for the same IP.
* This translating Hostname to its IP address is known as Name Resolution.
* Since as Network grew all these if an IP of host changed then we have to configure IP in all host files in all systems in order to solve this problem we moved all these hosts name associated with an ip into an single server in a network is called as Name Server to resolve al these hosts in a network.
* In /etc/resolv.conf you put IP of your Name Server server like:-

*nameserver 192.168.1.100*

and each time a system comes across a hostname which is not mentioned in the host file it.looks into this file and resolves the Hostname.

* If you have an hostname defined in the host file it first looks into that and then the Name server. If the name is mentioned in both a file is same then by default it first looks in /etc/host first but you can change that order by entry in file /etc/nsswitch.conf
* If you ping a host that is not in either list (/etc/hosts & /etc/resolv) eg [www.yolo.com](http://www.yolo.com) ?

you add a public -name server’s IP like 8.8.8.8 (google’s DNS) to resolve all the host your system doesn't know.

*nameserver 8.8.8.8 (Google’s Public DNS)*

* What is this www and .com in the end? → this is called a Domain Name to resolve public IP’s and the server they put in is called a DNS server.

. is root domain then com is domain then comes sub-domains.

* When you hit www.google.com it first goes through your internal domain name server of organization if it can’t resolve it goes to internet (a public DNS) then this DNS request goes to root domain then may be go through multiple domain and finally reach google’s IP then it forward it to you then if you hit any google’s subdomain like drive.google.com it resolves in google domain server etc. this how multiple resolves your hostname and you finally reach the site.Your browser may cache IP’s and request for faster communication.
* In /etc/resolv

search mycompany.com

here if you simply wanna ping web not web.mycompany.com, your system will automatically append mycompany.com and ping it.

* What is a Record?

mapping of a type e.g.

IPV4 to name mapping is called as A type

IPV6 to name mapping is called as AAAA type

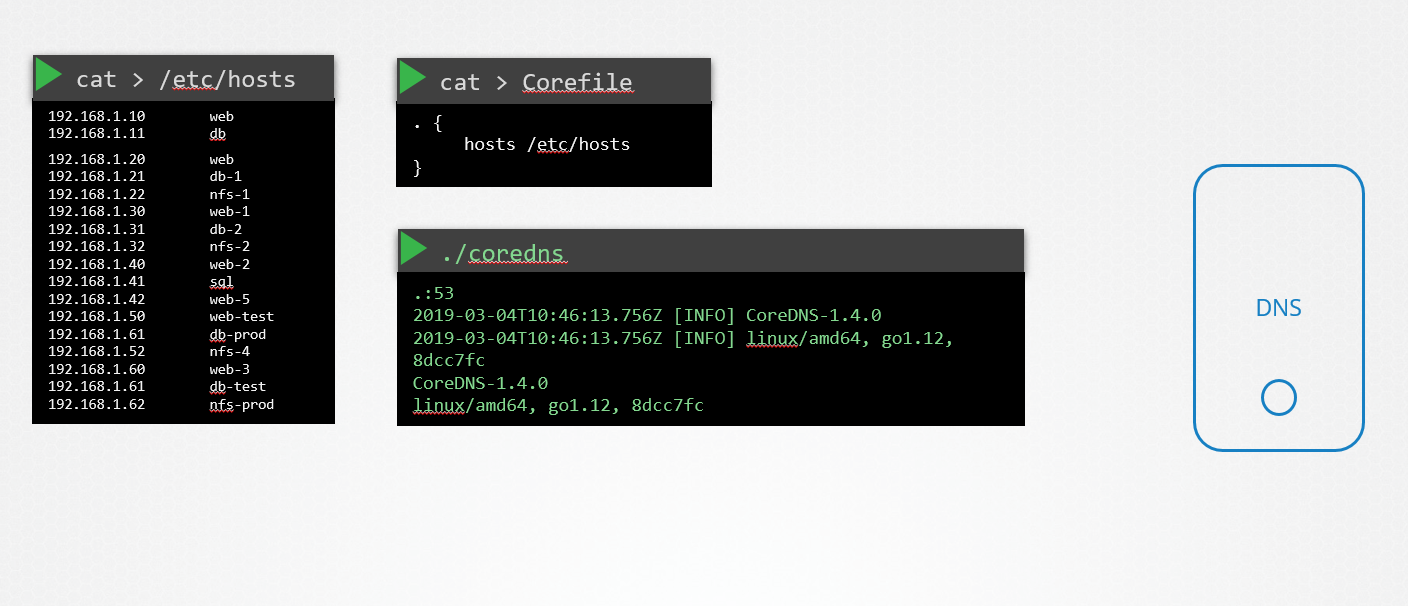
CNAME is name to name mapping

* Ping is not always good to test name resolution, use *nslookup* command. it queries a hostname from a DNS server nslookup does not consider entries in local /etc/host file.

Another is the *dig* command and has more details.

# CoreDNS:

* CoreDNS is a software you run on a DNS server and uses /etc/hosts file to configure hosts.
* It has a configuration from a file named Core File and is loads /etc.hosts file into it.



* CoreDNS also supports other ways of configuring DNS entries through plugins.and kubernetes uses a CoreDNS Plug-in.

# Network Namespaces:

* It is used to isolate different networks.
* Since the container is separated from the underlying host by namespaces.
* The container within its namespace have its own routing table, ARP table , interface etc
* Commands:-
* *ip netns red* to add network namespace
* *ip netns* to list namespaces
* *ip link* to list interfaces on host
* To establish connection between 2 namespaces by using virtual ethernet cable also known as pipe, therefore it has 2 interfaces for communication between (say) red and blue containers.
* Example:- Here blue and red are containers. veth-blue & veth-red are virtual cable names.

*ip link add veth-red type veth peer name veth-blue*

* attach each interface to appropriate to right namespaces:-

*ip link set veth-red netns red*

*ip link set veth-blue netns blue*

* assign ip to interfaces:-

*ip -n red addr add 192.168.15.1 veth-red*

*ip -n blue addr 192.168.15.2 veth-blue*

* up the two interfaces:

*ip -n red link set veth-red up*

*ip -n blue link set veth-blue up*

* But how do you do when you have lots of containers to connect? → create an internal network for containers. For creating a network you need a switch as a Virtual Switch for containers.
* There are lots of solutions available for virtual switches e.g. Linux Bridge etc.

we use Linux bridge so to create internal bridge network we use:

*ip link add v-net-0 type bridge* → From a host perspective it us just another adapter/interface you can see it by ip link command

*ip link set dev v-net-0* up→ to bring up the virtual switch

Think of it as an Interface for the host and a Virtual switch for the namespaces.

so connect the container namespace using this switch. we set up.

* Earlier red and blue containers are connected directly through virtual cable but now all others including red and blue will connect to this switch.
* so this virtual cable is of no use delete it→ *ip link del veth-red* other will automatically deleted here.
* now connect these virtual adapter/interface of namespaces to the virtual bridge network (just like in real world use connect ethernet cable to a switch network) by again creating cable for each container namespace and connecting interfaces of each namespace to this bridge network.
  + eg *ip link add veth-red type veth peer name veth-red-br*

*ip link add veth-blue type veth peer name veth-blue-br*

* one end of this interface attach to container namespace and attach other end of network switch use:-

ip link set veth-red netns red → this red virtual interface is attached to red namespace

ip link set veth-red-br master v-net-o → this red br virtual interface is attached to v-net-0 Switch now. and since we already linked to two interfaces now the red namespace is connected to the network.

* assign ip to rednamespace adapter→ ip link -n red addr add 192.168.15.1 dev veth-red
* up the red adapter→ iplink -n red link set veth-red up

The same procedure goes for blue and all other container namespaces.

now all are in the network and all can communicate with each other.

* If you want to establish connectivity between the host and these namespaces? → since for host this is network interface/adapter, assign an network addresses to this :

*ip addr add 192.168.15.5/24 dev v-net-0*

* you can ping any ip of the container now. But here the internal private network of these containers is still restricted. How do you connect this private network to the outside world→ say at eth0 interface of host the LAN is connected then→ add a gateway and this gateway is actually the ip of the network adapter that you have assigned to it: 192.168.15.5.

add a route entry in the blue namespace in order for blueconainer to communicate with outside world :-

*ip netns exec blue ip route add 192.168.1.0/24 via 192.168.15.5*

don’t need an additional route between eth0 and v-net-0 ..Have a NAT enabled to forward packet between internal private network from host to outside networks.

* You still don’t get the response back. The problem is the other host won’t know about you? you have sent him as a container ip its mac you have to masquerade as you are the machine and the other side will then recognize you. For this use NAT, the NAT will use its own ip and name to communicate and all the packets from the source will masquerade as NAT’s ip and name.

use:-

*iptables -t nat -A POSTROUTING -s 192.168.15.0/24 -j MASQUERADE*

* you can communicate with your LAN but to communicate with every network i.e. internet→ Add a Default Gateway if you want your container namespace to communicate with the internet.

use:-

*ip exec blue ip route add default via 192.168.15.5* → on blue container now can ping any network.

# Docker Networking:

* Docker has 3 types of networks mode.
* In None network mode the container has no network , it cannot talk to any other container and to the outside of the host.
* In Host networking mode it uses the Host network and there is no isolation between host network and docker network.so a port 80 uses by a container cannot be used by any other application on host.
* In Bridge Network mode Docker creates an internal private network just like the Linux virtual switch , it works the same like it creates virtual ethernet interfaces(docker0 usually with ip 172.17.0.1/24) and pipes between containers and the internalBridge network. For Port mapping internally it uses NAT.

# Container Networking Interface (CNI):

* Every Container solution solves this problem of Networking in the same way.
* Kubernetes also uses this approach.
* So this method of creating a switch, a virtual cable , pipes and then connecting them to the switch and thus creating a private network is called a Bridge.



* So why not a program called bridge is created which is used by all solutions to do all these steps but we need a certain standard used by every container solution and container orchester solution to adhere to these standard, this standard is called as CNI or container networking Interface and the bridge solution steps is a Networking solution which is used by weave-net, flannel etc like companies which uses these CNI standard in their solution.
* CNI are standard and procedures to how a network plug-in should be developed and how a container has certain methods to invoke networking through these plugins.
* CNI comes with a supported plug-ins already such as bridge, VLAN,IPVLAN, MacVLAN

etc and other plug-ins like weave, flannel, cilium etc.

* Here Docker does not adhere to this and has its own CNM so with kubernetes and 3rd

party plug-ins and docker with no CNI how does kubernetes integrate 3rd party plug in network with no support from docker? → kubernetes uses None network of Docker and creates a bridge network from one of the plug-ins it has.

# Cluster Networking:

* Networking required in Master and Worker Nodes in kubernetes cluster are as each hostname must be unique, each node must have at least one ip, have a unique MAC address etc
* There are some ports which must be opened as well they are used by different kubernetes components eg Master had kube-api which listens on 6443, kubelets listen on 10250 etc
* commands :-
* *ip link*
* *ip addr*
* *ip addr add ip-address-here*
* *ip route*
* *ip route add network(cidr) via gateway*
* *cat /proc/sys/net/ipv4/ip\_forward → must be set to 1 for forwarding.*
* *arp*
* *netstat -nlpt*

# Pod Networking:

* Earlier we set up networking between nodes enabling kubernetes control plane components to communicate.
* There is another layer of networking also i.e. the network that connects the pods together.
* How do pods communicate with each other ? an to external world
* Kubernetes does not come up with a solution and expects you to implement a networking solution for this.
* Kubernetes though layed out requirements for pod networking as:-
* every pod must have an IP address
* every pod should be able to communicate with every pod within the same node.
* every pod should be able to communicate with every pod on other nodes without NAT.
* We have a lot of networking solutions like flannel etc here the same networking namespace concept used earlier in Linux is applied the same concept basically applied everywhere.
* Steps to implement pod networking:-
* Assuming we have 3 nodes (1 master and 2 worker) these nodes are a part of external network and each node has its own IP say 192.168.1.11-13
* When containers are created kubernetes creates a network namespace for them and to enable communication attach these container namespaces to the virtual network → the bridge/virtual switch say vnet0 for node01 v-net1 & vnet-2 for other two nodes.
* Assign a IP address to the Bridge Network and interfaces → each bridge can have its own subnet ie any private address range will do say 10 .244.1.0/24 -3.0/24 and for the bridge interfaces → ip addr 10.244.1.1/24 dev v-net-0 same method to assign ip for other two.
* Now we have to attach each namespace container IP to this interface IP of switch and create a pipe between all of them, better to create a script for it.
* Create a virtual cable/pipe → create veth pairs→ ip link …...
* Attach one end to container namespace and another end to the bridge → ip link set …..
* Assign IP address → ip -n <namespace> addr add…...
* Add a route to the default gateway→ ip -n <namespace> route add …..
* Bring up the interfaces→ ip -n <namespace> link set…...

Let the script be net-script.sh and containes these commands run this script on each container.

The pods now have an ip address and are able to communicate with each other within their Nodes.

* Next part would be to enable pods in one node to reach other pods in other nodes as follows:-
* the ip of other pod is in different Network than the first pod → you already added the IP address of node01 as default gateway in each container in node01
* so in order to reach the pod in node02 the pod in system node01 knows it is in a different network so it forwards it to the default gateway which is actually the IP of node01 itself.
* But here the container in node01 (say blue pod) still cannot ping a container in node02 as node02 doesn’t know the IP of the container as it is an internal private network → add a route to internal network of node02 through node02 IP in node01 → ip route add 10.244.2.2 via 192.168.1.12

here 192.168.1.12 ip of node02 and 10.244.2.2 is one of the containers of node02.

* The blue pod is now able to ping to the node02’s pod. Now you have to do this config on all pods and nodes.
* Instead of doing all config of routes in each server it is better that we simply use a router and point all hosts to use that as the Default gateway, that way you can manage all routes in Router’s routing table.
* Now with the router the Network solution consists of 3 networks (10 .244.1.0/24 -3.0/24 i.e. 3 virtual switches) and all routes between them become one Big Network (10.244.0.0/16).
* For Network within Nodes we ran a net-script.sh script that performs all taks of routing and configuration but in large env thousand of pods are created every minute we cannot run script manually every time → here comes CNI into play acts as a middle man CNI tell this is when you call a script as soon as soon as container is created and this is how your script should look like(e.g. must have ADD section to ad all ips and DEL section).
* kubelet is responsible for container creation , it has a parameter **--cni-conf-dir=/etc/cni/net.d → find the path of script --cni-bin-dir=/etc/cni/bin→ executes script with container name & id ./net-script.sh add <container> <namespace>**
* A network solution like flannel will do these things automatically.

# CNI in kubernetes:

* How does kubernetes implement CNI and plug-ins?
* CNI defines responsibilities Of container run time→ Container runtime must invoke Network plug-in when a container is created, also when it is deleted,JSON format for network configuration.
* CNI pulg-in must be invoked by that component of kubernetes that is responsible for container creation and management i.e. Kubelet
* **In kubelet.service → --network-plug=cni; --cni-bin-dir=/opt/cni/bin; --cni-conf-dir=/etc/cni/net.d**

# CNI: Weave Plug-in

* Weave deploys its agent on each node.
* Each agent has the topology of the entire network that way they know where each pod, node etc is in the cluster.
* Weave creates its own bridge network on the node and then assigns the ip address to each node.
* Weave agent intercepts the sending packet , encapsulates it with its source and dest and sends it to the route network and at recieve end weave agent decapsulates the packet and sends it to the destination pod.
* Deploy Weave:-
* Weave & Weave agents can be deployed as services and daemons in the cluster manually.
* If kubernetes is set up already then it is an easier way to deploy as pods in the cluster.
* Once the Base Kubernetes cluster is ready with all pods and nodes configured and all network are configured then weave can be deployed directly as :-
* kubectl apply -f “https:// cloud.weave.works/k8s ...'' → this will instal all necessary components of weave automatically.

with this the weave peers/agents are deployed as daemon-set

* If the kubeadm tool is used then it is deployed as a pod in the kube-system namespace.
* For troubleshooting purpose view the logs of weave using:-
* Kubectl logs weave-net-56hdbf weave -n kube-system

# IP Address Management:

* IPAM for pod networking since it is a private network with its own subnets, IPAM for pods doesn’t concern Node's IP they can have completely different sets.
* How are the virtual bridge networks in nodes assign IP to subnets and their own.
* CNI has a built-in plug-in known as ‘host-local & which takes care of IPAM.’
* CNI configuration file (cat /etc/cni/net.d/net-script.conf) has a section called an ipam in which you can find plugins used for subnets and routes too.
* How does weave manage IP addresses? :-
* By default allocates Ip address in range 10.32.0.0/12 (10.32.0.1 to 10.47.255.254 ) → that's about a million IPs! for pods(2 power20 as 12 are net-id and remaining are hosts)
* These IP’s (10.32.0.1 to 10.47.255.254 ) are then divided equally between pods in different nodes and to every virtual switch/bridge

# Service Networking:

* You will always use a service to communicate to other pods not directly.
* When a service is created it is accessible to all pods irrespective of what nodes they are in.
* While a pod is hosted on a Node , a service is hosted across the cluster.
* The service which is only available within the cluster ( basically for pod internal communication) is called ClusterIP.
* The service which is available outside the cluster is called A NodePort Service.So in addition it exposes application ports in all nodes to outside the cluster.
* How are these services getting the IP address and How are the services made available across the cluster? :-
* Each node runs a kubelet pod for container creation and it get status of nodes through an api-server. Whenever a new container is created it invokes the --cni-plugin to configure networking for that pod.
* Kube-proxy also runs on every node and watches the changes through an api-server and each time a new service is created kube-proxy gets in action.
* Unlike pods services are not created on each node or assigned to each node, services are a cluster wide concept as they exist across all the nodes in the cluster.As a matter of fact they don’t exist at all there is server or service listening on the IP of the service.
* Service is a virtual object, How do we get an IP for service?How are we able to access the application on pod through services? → When you create a service object it is assigned an IP address from a predefined range. The kube proxy on each node gets that IP address and creates forwarding rules on each node in the cluster saying any traffic coming from the IP of the service goes to the IP of the pod.
* Remember not just the IP but also port forwarded with it. services as a combination IP and port.
* Whenever a service deletes the kube-proxy delete these rules.
* How are these rules created → kube-proxy use userspace, ipvs or IP tables etc . By default uses IP tables to do that.These proxy mode can be set while config kube-proxy service as:-
* kube-proxy --proxy-mode [userspace | iptable | ipvs]
* IP tables mode→when a service is created kube-apiserver assigns a IP address to it. the range can be set in apiserver.service parameter → --service-cluster-ip-range ipNet by default it is set to 10.0.0.0/24
* Here you should not overlap ranges of cidr of services and pods (If configure manually). assign them different ranges. That’s how an IP service is assigned to a service, here you can check in system by:-

*iptables -L -t net | grep db-service*

* you can see in kube-proxy log → *cat /etc/var/log/kube-proxy.log*

# DNS in kubernetes:

* DNS resolution of the kubernetes cluster nodes already is like having a DNS server on your organisation and nodes with IP’s associated with names.
* Here we are talking about DNS within the Kubernetes cluster between name resolution of pods and other objects of the cluster. Kubernetes deploys a built-in DNS internally by default when you set up the kubernetes cluster.
* If we set up kubernetes manually then we do this by ourselves.
* Whenever a service is created the kubernetes DNS creates a record for the service and it maps the name of the service with IP address of the service.Now any pod can reach this service using its service name.
* If they are in the same namespace using just the name is enough to reach it. If it is another namespace use as: name-of-service.namespace-name
* For each service DNS creates a separate subdomain with the name of namespace→ web-apps.web-apps-namespace. All services have another sub-domain called svc to group them (type sun domain for all types of object) → web-apps.web-apps-namespace.svc
* Finally all the services and pods are group together into a root domain for the cluster which is set to cluster.local by default → web-apps.web-apps-namespace.svc.cluster.local
  + *curl http://web-apps.web-apps-namespace.svc.cluster.local*
* For pod the records are not created by default but you can enable it explicitly.All works the same just the name of the pod given is IP address with ‘-’ between them.

# CoreDNS:

* Since we need a DNS server inorder to resolve names associated with ip whether of pods, services etc.
* Kubernetes deploys a DNS server within the cluster called CoreDNS.
* The CoreDNS is deployed as a pod in the kube-system namespace.Actually deployed as 2 pods for redundancy.
* -This pod runs a CoreDNS executable.
* It requires a config file as Core file→ cat /etc/coredns/Corefile, within this file a number of Plug-in are configured. It is where the top level domain cluster.local is configured.
* pods option/parameter in Core file is responsible to enable DNS for pods. By default it is insecure mode meaning it is not enabled.
* Any name the CoreDNS can’t resolve is forwarded to /etc/resolv.conf.
* This Core File is passed to the pods as ConfigMap object as:
* kubectl get configmap -n kube system
* You can edit this configmap that way if you need to modify the configuration you can edit this config map.
* It watches for new pods or services and adds records of service in its database.
* What address does the pods use to reach the DNS server? → when kubernetes deploys a CoreDNS pod it also deploys a service kube-dns and the IP address of this service is configured as a nameserver on the pod's env in /etc/resolv.conf file. This is done automatically and it is done by kubelet. In the kubelet config file (/etc/lib/kubelet/config.yaml) you see IP of DNS & domain (cluster.local) in this file.
* You don’t need to specify full domain name as *web-apps.web-apps-namespace.svc.cluster.local*  because resolv.conf file has a search entry with .svc.cluster.locale , .service.cluster.local, cluster.local etc.
* host web-service

# Ingress:

* A Load Balancer is needed for a proxy server every time a new service is created to redirect the traffic and each of this cost.
* A lot of configuration of IP, LBs,https(SSL) security implementation, DNS etc are needed whenever you decide to deploy a new application and services on the same cluster.
* This increase cost and management of your kubernetes cluster that's where Ingress comes into play
* Ingress is like a virtual LB built-in kubernetes and provides a single accessible url to all your services based on the url path. It routes traffic to different services and configures SSL security.
* Ingress is just like any other object in kubernetes. Even with ingress you still need to expose your application to make it accessible→ publish as Nodeport or LoadBalancer for cloudnative apps.
* All Load Balancing configurations, SSL ,Routing based config , yrls etc on the Ingress controller.
* If not ingress then we would use a reverse proxy like HA proxy or nginx etc. Deploy them in clusters and configure them to route traffic to other services.Ingress kinda works the same way.
* In Ingress we set solutions for these reverse proxy and configure it.The solution we deploy is called an Ingress Controller. And the set of rules we configure are called Ingress Resources.
* We create Ingress Resources as definitions files like pods , rs etc. A kubernetes set up does not come with an Ingress controller by default.
* -Ingress controller we have Nginx, https 7 layer-LB of google , HA proxy etc Nginx and https of google are supported by kubernetes.
* An Nginx controller is deployed as just another deployment in kubernetes. THis controller made for kubernetes had its own set of options and arguments to be passed in the pod definition file → pass a config map of nginx to this pod.
* Expose this Nginx pod as a service Node port with port numbers 80 and 443. It also needs a Service account for special permission of its Roles and RoleBindings.
* Now comes creating Ingress Resources, they are set of rules and configurations applied on Ingress controller like simply forward all traffic to a single url etc.Traffic routing is controlled by rules defined on the Ingress resource.

-Ingress Resource YAML format:-

**apiVersion**: networking.k8s.io/v1beta1

**kind**: Ingress

**metadata**:

**name**: simple-fanout-example

**annotations**:

**nginx.ingress.kubernetes.io/rewrite-target**: /

**spec**:

**rules**:

- **host**: foo.bar.com #if you leave this field empty then it is applied to all hosts

**http**:

**paths**:

- **path**: /foo

**backend**:

**serviceName**: service1

**servicePort**: 4200

- **path**: /bar

**backend**:

**serviceName**: service2

**servicePort**: 8080

-For 2 Domain name we pass like this:-

**apiVersion**: networking.k8s.io/v1beta1

**kind**: Ingress

**metadata**:

**name**: name-virtual-host-ingress

**spec**:

**rules**:

- **host**: foo.bar.com

**http**:

**paths**:

- **backend**:

**serviceName**: service1

**servicePort**: 80

- **host**: bar.foo.com

**http**:

**paths**:

- **backend**:

**serviceName**: service2

**servicePort**: 80

* Command:-
* kubectl describe ingress ingress-wear-watch

## Kubernetes “The Hard Way”:

* Design a K8 cluster HA cluster etc.; Kelcy Higtower ‘s Kubernetes “The Hard Way”; end to end test to validate the cluster.

# Design a Kubernetes Cluster:

* Purpose:-
* For Learning purpose→ kubeadm or minikube or GCP or AWS
* For Development & Testing→ Multi-node cluster with single master and multiple worker nodes using kubeadm or GCP OR AKS or AWS
* For Production applications→ High Availability Multi-node cluster with multiple master nodes using kubeadm, GCP, kops, AWS etc. ; have upto 5k nodes, 150k pods, 300k container, 100 pods per node
* Cloud or On Prem? → use kubeadm for On prem and the cloud services of azure, google GCP etc for cloud.
* Storage? →
* High performance workloads relay on SSD backed storage.
* Multiple Concurrent connections use Network based storage.
* Persistent shared volumes for shared access across multiple pods.
* Nodes? → VM or Physical Machine; Min. 4 Nodes cluster; use 64 bit linux architecture.
* You can separate the ETCD clusters from master nodes to its own cluster.

# Kubernetes Infrastructure:

* Kubernetes can be deployed in various ways.
* For production env 2 solutions to deploy:-
* Turnkey Solutions→ you provision VMs; you configure VMs; Scripts to deploy cluster; You maintain VMs yourself e.g. kubernetes on AWS using KOPS.
* Hosted Solutions→ like a kubernetes -as-service thing, all things done by the provider e.g. Google Container Engine lets deploy a k8 cluster very easily.
* Turnkey solutions like Openshift by RedHat, Cloud Foundry Container Runtime, VMware cloud PKS etc. to deploy and manage kubernetes and configure them.
* Hosted solutions like Google Container Engine, Azure kubernetes service, AWS EKS etc

# Configure in HA:-

* Master Node can fail so better a redundancy. Therefore configure Master in High Availability by adding more than one node, e.g. with 2 node master.
* The api-server component of Master Node works in active-active mode while other in active passive mode.
* Put a load balancer for this to distribute the load between these 2 active api-servers.
* Scheduler n controller in active passive mode with leader options set.
* Kube controller endpoint object is locked by any process to become the leader and do tasks.this object is leased for a time and again both components retries after a few seconds.
* etcd also should have a separate server for redundancy so if master fails we can restore it from etcd.

# ETCD in HA:

* ETCD is a distributed reliable key-value store that is simple, secure & fast.
* In key-value stores each record has its own file in key-value format.
* For this data to become large and complex we use Json or YAML.
* We store ETCD databases in different servers for redundancy. Since it is distributed it maintains an identical copy in all redundant nodes.
* Data Consistency:-You can write in any instance and read from any instance from any of these nodes. For read since all servers have the same data you can read it & for write ETCD gives only one node permission to write through a leader elect process internally. If the write request comes through the leader it processes the request and makes sure all have the same copies and if write came through a different node i.e. a follower then it forwards the request to the leader elected.
* Leader election:- use RAFT protocol.
* The write request is said to be complete if the ‘majority’ of the nodes the write request is processed i.e. data is written on these.This is to provide a fault tolerance of node failure. this is calculated by :- Quorum=N/2+1 is the total no of majority of nodes.
* Recommended to always use a Odd no of nodes to meet the quorum so that in case of network failure nodes don’t evenly divide and we lose the quorum to cause the whole cluster failure.
* To install the ETCD :-
* Download the Latest supported binary→ wget --https-only “https://github.com /…”
* extract it → tar -xvf etcd-v3…
* Create the right directory structure→ mv etcd.v3.. /var/lib; mkdir -p /etc/etcd /var/lib/etcd
* copy over the generated certificates required for ETCD→ cp ca.pem 8key.pem k8.pem /etc/etc/etcd
* Then configure the etcd service→ ExecStart=/usr/local/binetcd --name= --cert-file= --key-file= …..--initial-cluster peer-1= peer-1\_ip:2380; peer-2=peer-2\_ip:2380
* Use ETCDCTL utility (Use Version by set env variable to 3 as by default it is 2)--> ETCDCTL\_API=3; etcdctl put name john; etcdctl get --keys-only

-So For a HA set up and the right fault tolerance and cost Nodes=5 is a good choice beyond it is unnecessary and below it is we get low tolerance with 3 nodes remember not to include even number of nodes.

## K8 Cluster Testing:

# End to end test:

* Manual Test:-
* check nodes→ kubectl get nodes
* check pods → kubectl get pods --all-namespaces
* check control plane components → kubectl get pods -n kube-system
* If they deployed as a service then→ service kube-apiserver status; service kube-proxy status etc.
* check all objects , check networking etc.
* Instead of testing manually use a kube-test suite, it has over 1000 test to check functionality of kubernetes.

eg test if intra pod communication is functional

* This test infra can build-deploy-test-clean all processes and check your kubernetes solution.
* If you have a test solution then you should pass a 160 Conformance Test in order your solution is kubernetes certified.

# Run E2E test:

* Run the go get -u k8s.io/test-infra/kubetest on Master node(have go line installed).
* kubetest --extract=v1.11.3 download the binaries with relevant versions of k8s.
* This creates a directory named kubernetes cd kubernetes.
* Run kubetest --test --provider=skeleton > test.txt
* Skeleton option describes a local cluster and also provide 2 env variable if skeleton keyword is used:-
* export KUBE\_MASTER\_IP=”192.168.1.4”
* export KUBE\_MASTER=kube-master
* During this test additional memory is required and this test need not necessarily run in master node you can run it from any linux system to test the k8s cluster.

# Troubleshooting K8s:

# Application Failure:

* Draw a Map of application structure like Db→ Db-service→ web→ web-service
* Start from either end to test application eg
* Try curl the front end part of application eg curl [http://web-service-ip:node-port](about:blank).
* Check connectivity of end points eg from service to pod by kubectl describe service web-service
* Compare the labels and selectros on service to the pods.
* Check status of pod kubectl get pod and also check restart counts.
* Check events of pods by kubectl describe pod and logs of the by kubectl logs.

# Control plane failure:

* Check control plane by kubectl get pods -n kube-system or check services if running directly service kube-scheduler-master status etc
* Check logs of these pods by kubectl logs kube-scheduler-master
* Check events in pods by kubectl describe pods kube-scheduler-master -n kube-system
* Correct the right parameter eg may wrong conf file name or path , might be wrings certificates name, path, exec commands check etc
* Tip:-
* Control Plane components in kube-adm runs as static pod and **node-name is appended whener a static pod is deployed in node** in this case as kube-scheduler-**master**.
* since in the kubeadm tool it **runs these as a static pod using kubelet** therefore it is good to **check the kubelet service process status and config files** eg

cat /etc/systemd/system/kubelet.service.d/10kubeadm.conf ← check this.

go to config path usually→ /var/lib/kubelet/config.YAML

check parameter staticPodPath

it is usually→ /etc/kubernetes/manifest/

# Worker Node Failure:

* Check by:-
  + kubectl get nodes
  + kubectl describe node node01
* Now there are 4 types of failure and a status is set to true if it is true.

eg MemoryPressure True → means node is out of memory etc.

* The status is unknown if the master lost connectivity to the node.
  + -checks df -kh, top, etc go to node and fire this.
  + -Check status of kubelet by service kubelet service status , check logs of kubelet service.
  + -check the kubelet certificate if they are expired, issued by right ca etc.
  + -check if connection between api-server and kubelet is okay.
* use kubectl cluster info
* when you change service arguments use→ systemctl reload daemon or,
* you can simply restart the service → systemctl kubelet.service restart

# Kubernetes Advance commands using Json Path:

- apu-server gives the data in JSON format and kubectl edits it in a readable format for us.

-to get started follow:-

* use kubectl get <object-name> -o=jason
* Make a json query for the data you want
* put json query in the kubectl command → kubectl get nodes -o=jason=’{items[0].metadata.name}’
* here items have all the object and items[\*] will give info about all the objects of k8s and n need to append the root $.
* it also has a sortby, loop & a custom column function eg:-
* kubectl get nodes -o=jsonpath=’{.items[\*].metadata.name}{“\n”}{.items[\*].status.capacity.cpu} ’
* kubectl get nodes -o=jsonpath=’{range.items[\*]}{.metadata.name}{“\t”}{.status.capacity.cpu} ’
* kubectl get nodes -o=jsonpath=’{.items[\*].metadata.name}{“\n”}{.items[\*].status.capacity.cpu} ’
* kubectl get nodes -o=custom-column= Node:.metadata.name,CPU:.status.capacity.cpu ’
* kubectl get pv --sort-by=.spec.capacity
* kubectl get pv -o=custom-columns=NODE:.metadata.name,CAPACITY:.spec.capacity.storage --sort-by=.spec.capacity.storage > /opt/outputs/pv-and-capacity-sort