**KUBERNETES (k/8’s)**

Since we have a lot of containers running in server/node. It is very difficult to manage and maintain the containers. There is an orchestration, tool for containers by which we can manage

You have one docker engine, if that docker engine crashes your entire containers are gone so we go for Kubernetes

Orchestration tools:-

1. Kubernetes (Google)
2. Open shift (RedHat)
3. Docker swarm (Docker)
4. Apache mesos (Apache)

Kubernetes is a container orchestration tool with which we can deploy, scale, monitor, and manage containers

(or)

Kubernetes is for managing containerized application in a clustered environment

* A Kubernetes cluster can have multiple nodes and multiple masters for high availability.

Why k8:-

1. Cluster of containers
2. Deploying application
3. Scale application when needed

Benfits:-

1. Portable
2. Extensible
3. self healing

Cons:-

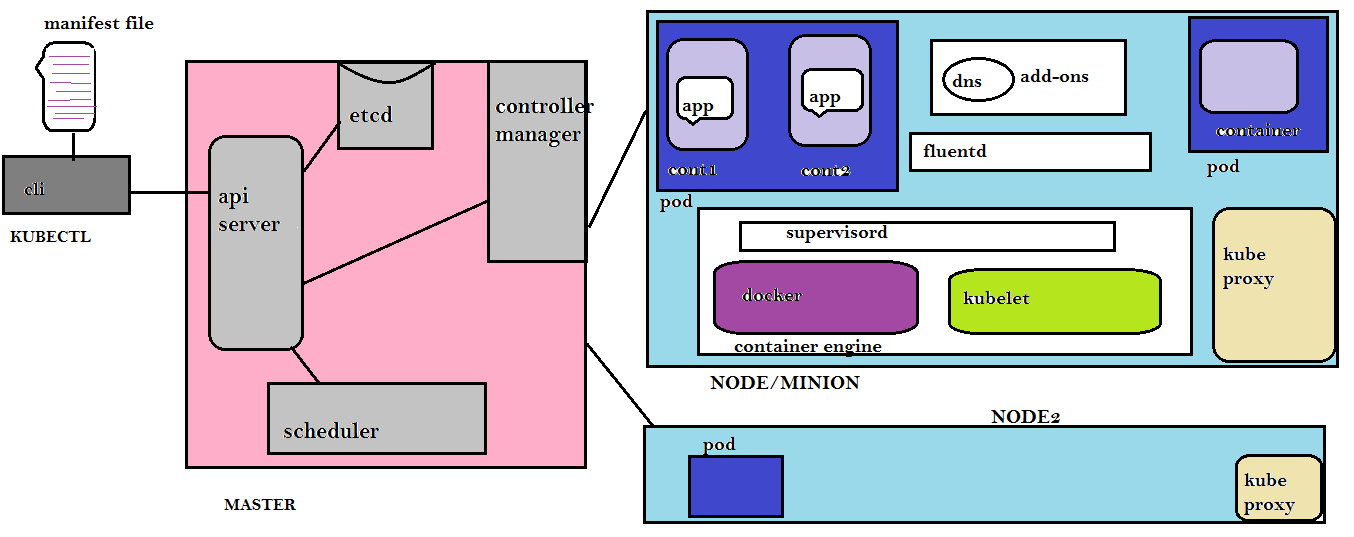
1. complex networking

Declarative vs imperative configuration:-

Declarative configuration means you tell system what to do and system will take care of it. Ex:- you write a yaml file for rc, that you want 5 rc. When 1 pod is killed system will automatically create a new one

Imperative configuration is where user will perform action. Ex:-starting 5 replicas of web container

KUBERNETES ARCHITECTURE:-



KUBECTL:-

Kubectl is a command line tool which communicates with api server and sends command to k8 master

KUBERNETES MASTER:-

The Kubernetes master runs api server, scheduler, control manager, etcd. All the components in master are called control plane.

Here we see single master server but in production we have multiple masters to support high availability

1. Api server:-

It is a key component in Kubernetes master where all components in master will interact with api server. Api server is like a watchman/entrypoint.

* Api server is brain of Kubernetes cluster
* Api server is responsible for authorization and authentication (all clients must be validated inorder to interact with api server)
* Api server implements watch mechanism for clients to watch if there are any changes

1. Scheduler:-

If there are newly created pods not assigned to any node, then scheduler responsibility is to assign the newly created pods to nodes

Scheduler also distributes workloads and track workload utilization on cluster nodes

1. Control manager:-

Control manager is a daemon. There are many controllers like

1. replica sets

2. deployments

3. statefulsets

4. daemon sets

5. jobs(ontime)

6. cronjobs

1. Node controller:- responsible for noticing and responding when node goes down
2. Replication controller: - assume if there are 4 pods running in a node and unexpectedly one pod killed. So replication controller maintains desired state (4 pods). Replica set is the latest name for replication container with minor changes. So some also calls RCv2
3. Endpoint controller:- takes care of relationship between services and pods
4. Namespace controller:-
5. Deployment controller:-

\*control manager duty is to watch the state of cluster. Whenever a state is changed then control manager will maintain desired state

1. Etcd:-

It is lightweight database which stores cluster information. where it maintains all the information like (which pods are running in which nodes, what node exists in the cluster, pods state, service details, namespace, replication information, networking etc…. at any given point of time). It is used to store as distributed key-value store and it is also a service discovery tool All k8 objects like pod, deployment, service... are created in k8 cluster are stored in Etcd

\*we must have backup of etcd otherwise Kubernetes cluster is gone

WORKFLOW:-

1. Kubectl writes to the api server
2. The api server checks in etcd and validates and store the entries in etcd
3. Now etcd notifies back to api server
4. Now api server invokes scheduler
5. Scheduler decides where to run the pods and returns to api server
6. Api server stores it in etcd and etcd notify back to api server
7. Api server invokes kubelet in the corresponding node
8. Kubelet talks to the docker daemon, to create container
9. Kubelet updates the pods status to api server
10. Api server stores the new state in etcd

KUBERNETES NODE:-

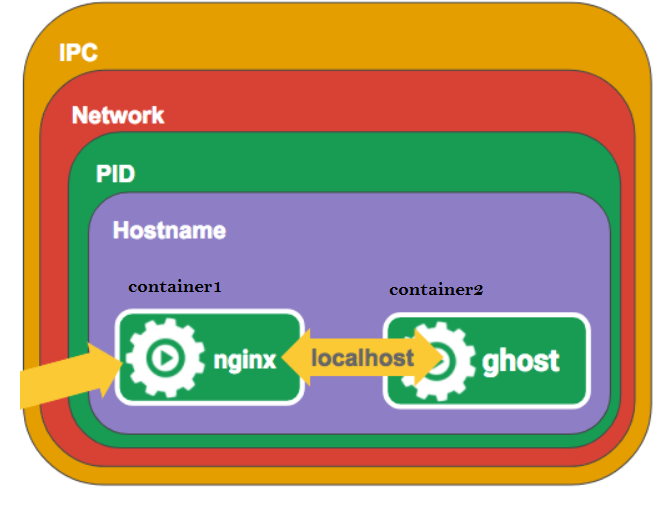
A node is also called as worker node or minion or slave. A node can run physical machine or virtual machine but runs only on Linux o.s. Imagine if the entire node (which contains containers inside it) goes down immediately a new node comes up

1. Pods:-

Pods are smallest deployment unit in Kubernetes(we can create only pods but not anything smaller than that). A pod is a place where one or more containers run inside. Here pods will have one ipaddress no matter how many containers run inside it. All containers inside a pod will share pod resources like storage, networking, volumes, filesystem…

* pod can talk to each other using ip address and ports

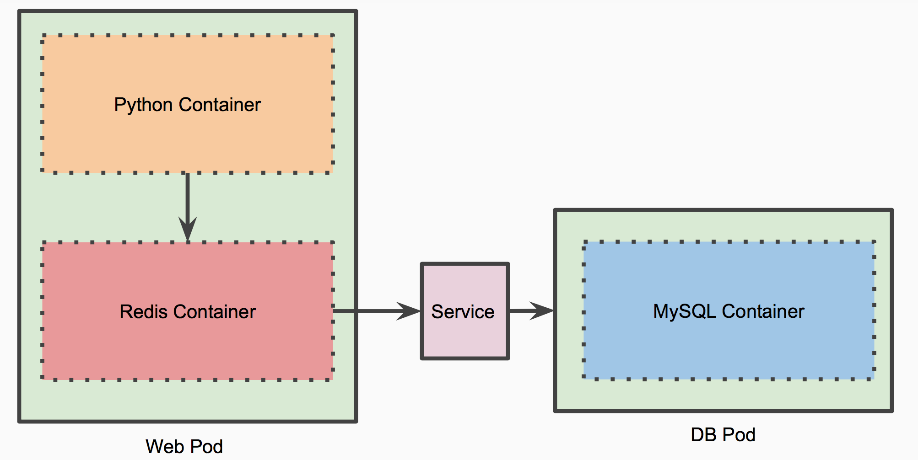
why pod why not container. If you are building everything inside a container, the purpose of container differs and what if we want to monitor the container? So is the reason we go for pod which have containers running inside them



Pod lifecycle (i) pending (ii) running (iii) succeed/failed (iiii) unknown

A pod has a section called restart policy. Restart policy means restarting the containers by kubelet on the same node. Exited containers will be restarted with some time delay

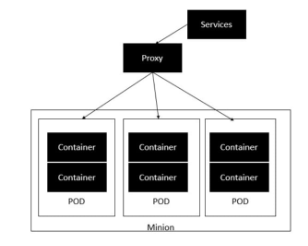
1. always:- this is default value. If container exits with success(zero exit code) then restarts the container
2. onfailure:-restarts the container if container exited with failure(non-zero exit code)
3. never:- it means container will never restart irrespective of why it exited



1. I have web pod which has python flask container and redis container
2. I have one more pod where mysql container is running
3. whenever data is retrived through python, it should first check redis before accessing mysql
4. each time data is fetched from mysql , the data gets cached in redis container and python flask will access the data
5. when new replica set is created for web pod, the python container and redis container are scdeduled together
6. Kube-proxy:-

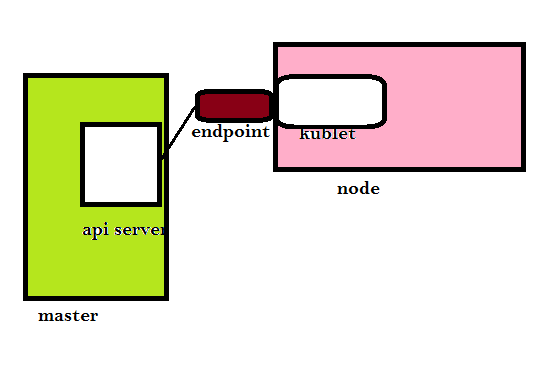
This is networking component which runs on each node. It make sure that communication between pods, communication between containers, communication between nodes, communication between pods and nodes, communication between clients and services are happening and also kube-proxy ensures networking environment is isolated

Kube-proxy allows network traffic to redirect to various pod.



1. Kubelet:-

Kubelet is a dumb agent that runs on each node and ensures weather all pods and containers are running in a node at any given time. It reports to api server if a pod state is changed (if pod dies/unhealthy). Then api server will invoke scheduler to assign new pod. (kublet will check if container is healthy or not. if container is unhealthy kublet kills container) Actually control manager tracks node status through kublet agent



Where end point is a network interface

1. cAdvisor:-

By default cAdvisor is present in kubelet. cAdvisor auto-discovers all containers in the given node and collects cpu, memory, network,

filesystem usage

1. Container engine:-

Every node contains container runtime interface. It is responsible for starting and managing containers. Container engine is pluggable means you can run docker engine now, but in future if you want to run rocket you can run it

1. Supervisord:-

Both docker and kubelet are packaged in a layer called as supervisord. Supervisor ensures docker and kubelet are running all the time

1. Fluentd:- It is a daemon which is responsible for managing logs and talking to central logging mechanism
2. Add-ons:-

Add-ons are optional like dns, ui etc..

1. Endpoint: - services doesn’t link to pod directly instead there is a resource which sits between service and pods ie.., endpoint. Endpoint is a list of ip and ports exposing to service.

Whenever you create a service, endpoint is automatically created. Ex: - if you have 3 pods when you create a service then 3 endpoints will be created

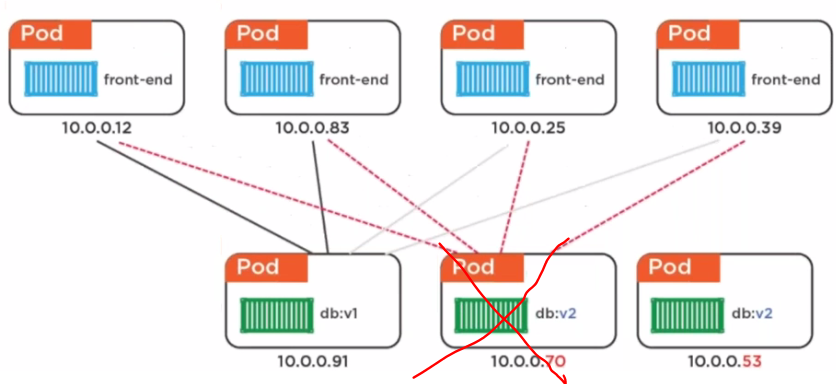
* Endpoint will be manually created when you don’t define selector in a service
* Whenever a pod in a service changes, endpoints will be updated

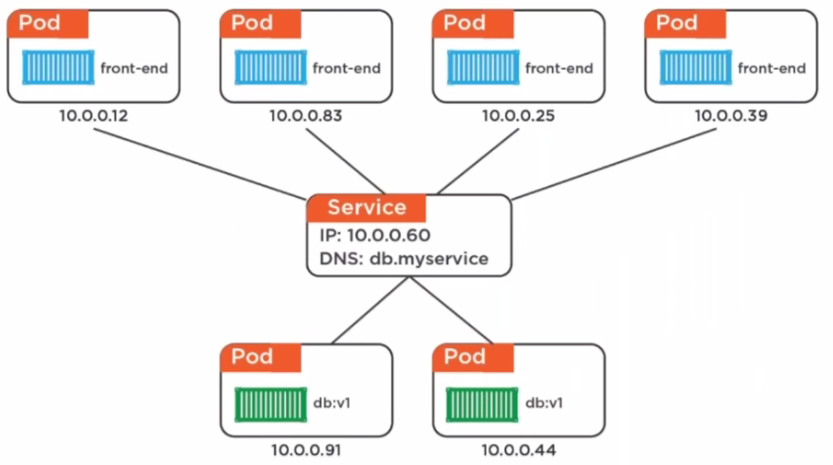
1. Resources/objects: - Object means at any given point of time they will be running. These Kubernetes resources will be defined in yaml file but executed at runtime. Some of them are
2. secrets
3. pods
4. deployments
5. replica sets/replication controller
6. services
7. ingress
8. namespace
9. config maps
10. persistent volume claim
11. end point
12. resource quota
13. job
14. Services:-

Earlier I said each pod is allocated with one ip address. Imagine if one pod is killed, so a new pod will be created with new ip address. So communication doesn’t happen. Moreover a pod is never exposed to the outside world. We expose pods through services

In order to avoid all these things services comes into pictures. We write all ip address, ports, dns, etc... to services. So service does load balancing to all the pods by selector, labels.

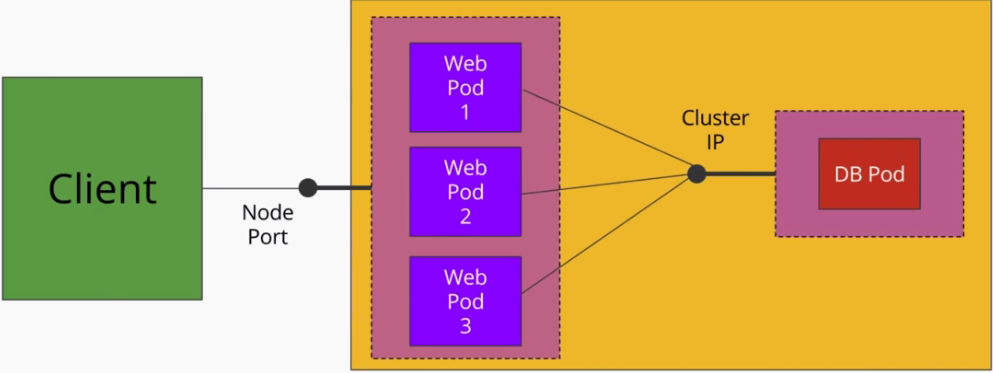
* services checks if pods are healthy or not and does load balancing only to healthy pods
* every service has virtual ip, port, dns name which remain constant throughout the service lifetime
* services communicate each other with names
* every service has an virtual ip address (also called cluster-ip)





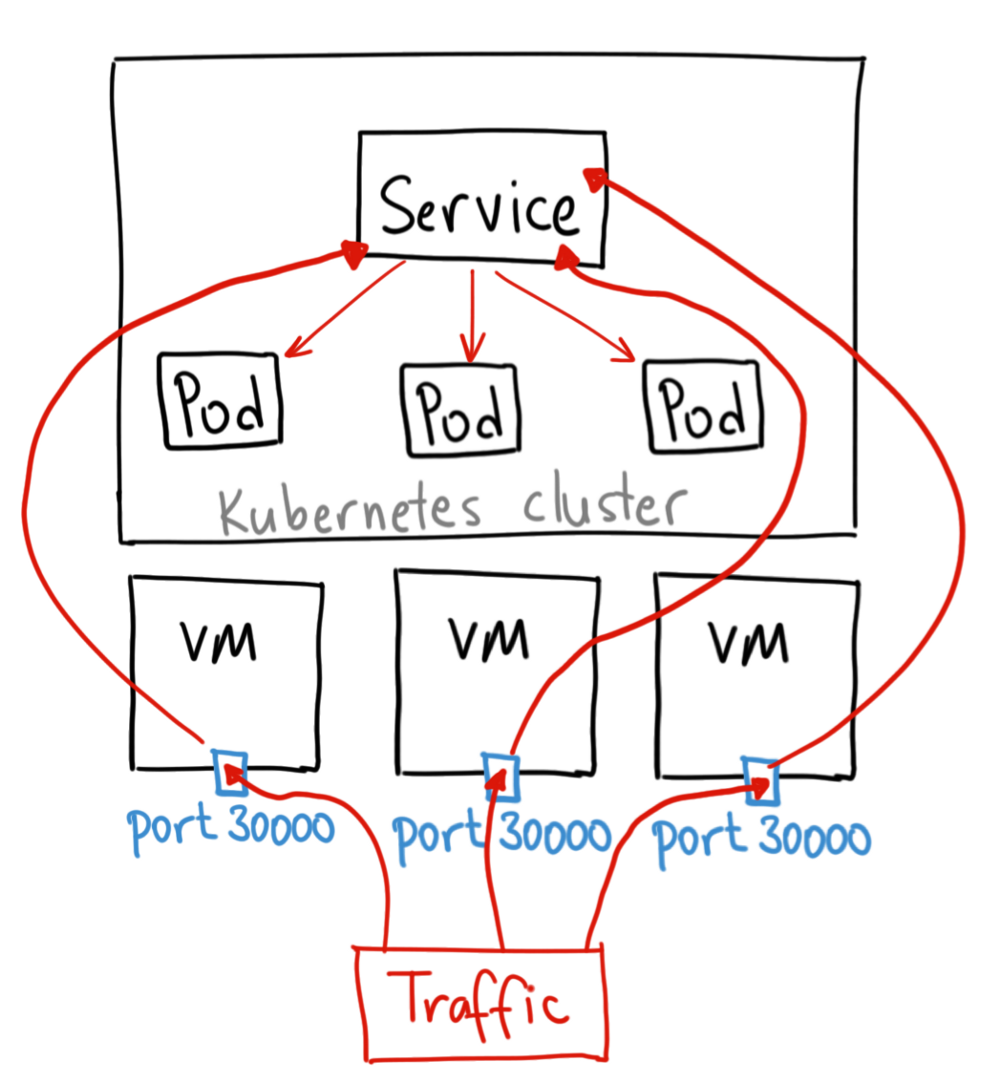
We can expose services in 3 different ways

1. clusterip:- exposing a service on an internal ip within the clusters,so that apps inside the cluster can only access



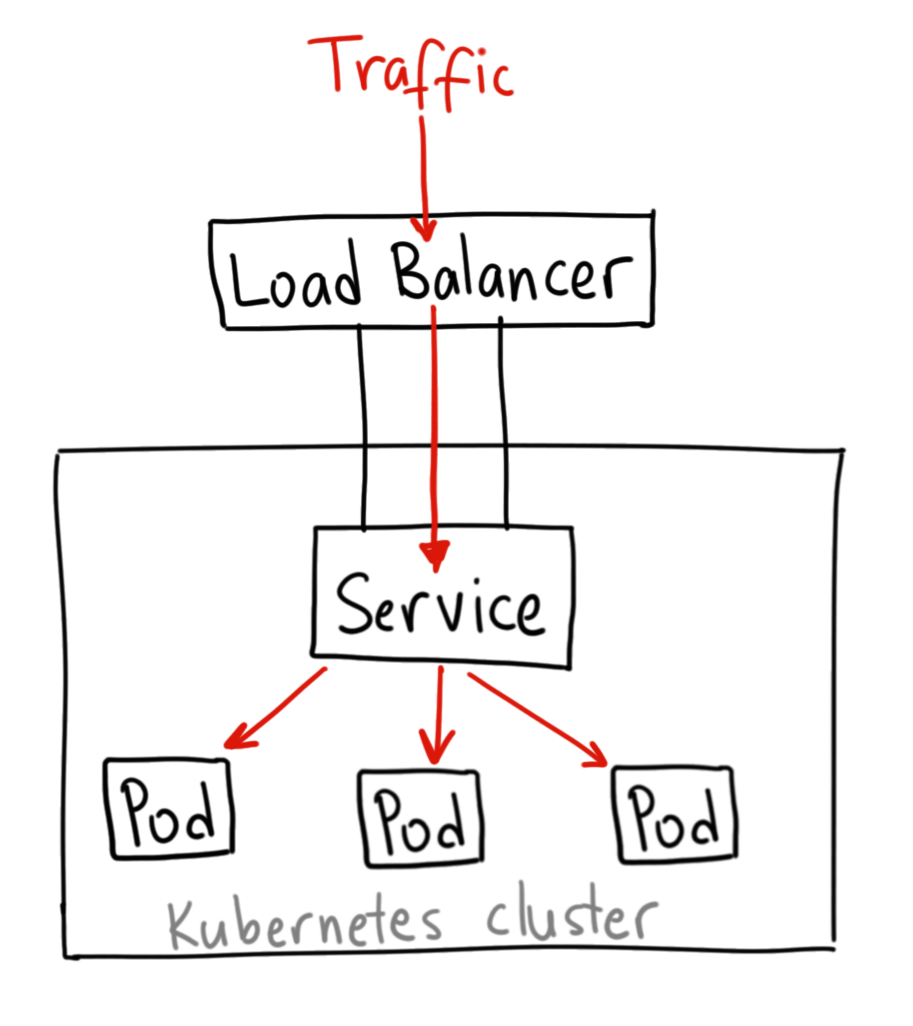
Ex:- frontend pods want to communicate with backend pods through a service called cluster ip.

1. Node port:- exposing a service on same port of all worker nodes. The service can be accessed from outside the cluster using the NodeIP:nodePort



Ex:- if external world has to access, then the incoming request hits node ip address on the specified port then gets routed to particular service

1. Load Balancer: - exposes your services to the external world through cloud load balancer. the major drawback is every service that is exposed through load balancer will get an ip address and is chargeable



Port: -Expose the service on the specified port internally within the cluster

Target port: -target port is a port on the pod. Services will route traffic to this port. Target port can be number or a string

Container port: - the port on the container

1. Service discovery:-

The process of pods finding other services running on the same cluster is called service discovery

Imagine if you are upgrading/auto scaling/node failed, then service will dynamically launch in new network location hence it is difficult to track all the services. So service discovery comes into picture

* Service discovery dynamically discovers where the service is?
* Service discovery will be used as service registry, health checker, load balancer, dns server for an application

We have 2 types of service discovery

1. Environmental variables:-

When a pod is created the kubelet adds some set of environment variables to the container for each active service

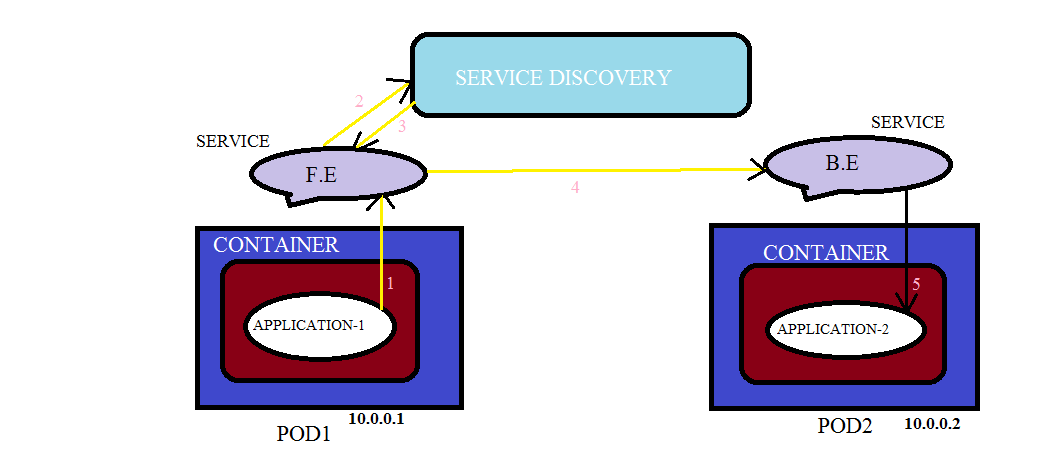
Ex: - there is an nginx service and you created pod the kubelet adds environment variables to the container, this can be seen by following command

#kubectl exec PodName env

1. DNS:-

DNS is an add-on and strongly recommended to use. Each node has local DNS server. This DNS server watches api server for new services and creates a set of DNS record for each like <service-name>.<namespace-name>.svc.cluster.local. Then the pod can do name resolution of service automatically.

Ex: - if you create an nginx service in default namespace then a DNS record “nginx .default” is created



Assume we have a web application running inside container which is in pod1 and there is service FE. Similarly we have db application in a container which is in pod2 and there is a service BE.

Whenever a service “FE” and “B.E” is created in production namespace automatically DNS server creates a record “FE.production.svc.cluster.local” and “BE.production.svc.cluster.local”.

There is pod1 which has service “FE” in default namespace similarly you have pod2 which has a service called “BE” in default namespace. Pod1can lookup service “BE” by simply doing nslookup BE. But assume if service “FE” is in default namespace and “BE” is in prod namespace then pod1 wants to access service “BE” it can do a nslookup by BE.prod Now the pod1 has ip 10.0.0.1 and pod2 has ip 10.0.0.2. If pod1 wants to access pod2. It will access using service name if they are in same namespace

DNS:-

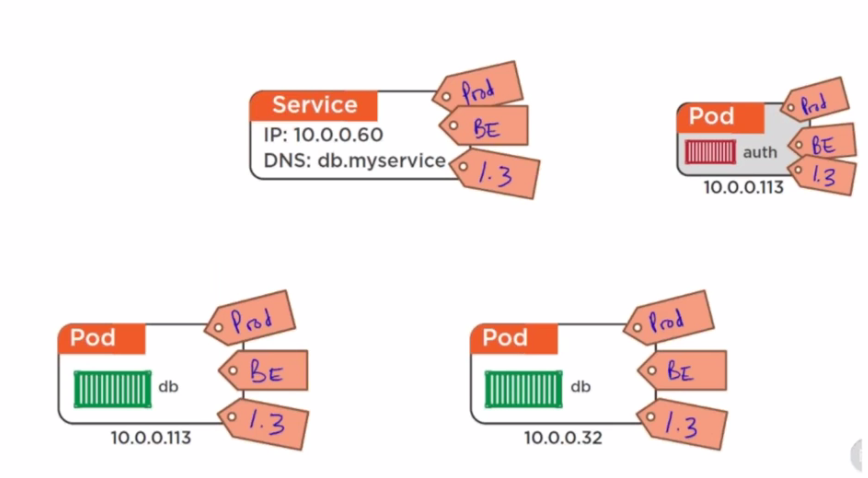
One service will connect to other service by name and ip address will be resolved at runtime. So how this is happening? Using dns (we connect to Google by domain name but backend its connecting to ip address). Also a service for dns is created if you see /etc/resolve then this file points to internal dns (kube-dns). kubernetes master automatically update kube-dns whenever a service FE queries dns server then it replies back service BE ip

SERVICE DISCOVERY TOOLS:-

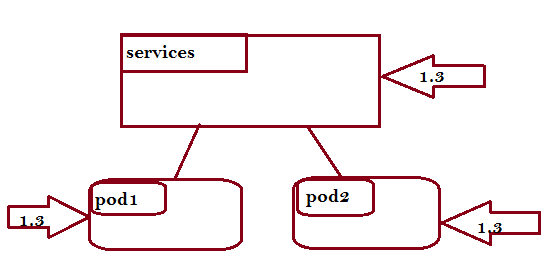
1. Etcd-----core o.s
2. Consul-----hashicorp
3. Zookeeper-----apache
4. Eureka------netflix
5. Labels:-

Labels are key value pair or name tags, which are attached to Kubernetes objects (pods, replication controller, services). Just like how we add tags to ec2 servers to identify them

Services tries to load balance based on the pods label

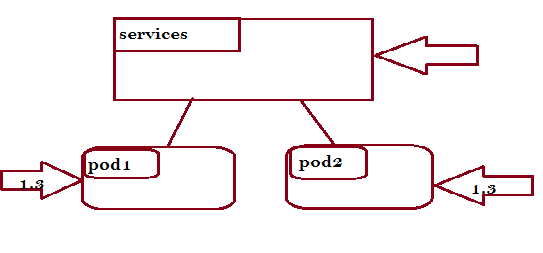


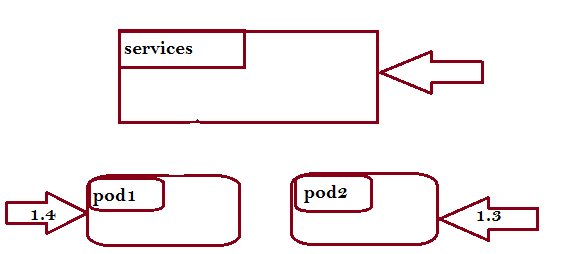
Here labels for service is (prod, b.e, 1.3). If service labels and pod labels matches then service will do load balancing to those pods. Service doesn’t have brain to find whether the pods are running backend or frontend, it does load balancing only using labels

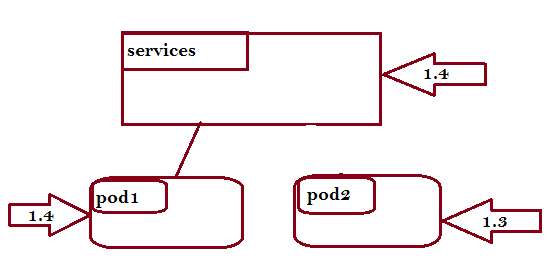


If you want to update the pods to 1.4 then

1. remove 1.3 label in service
2. now bring up pods with 1.4 label
3. change the label to service 1.4
4. now service load balances to 1.4 pods







1. Selectors:- The kubernetes objects uses selectors to identify the pods.

Ex: - if service has to load balance pods then service object uses selector, this selector and pod label should match

* There are 2 types of selectors

1. **Equality-Based Selectors:-**

This selector will filter objects based on label keys and values ex:- env==dev

* Here we can use operators like ==, !=,

1. **Set-Based Selectors:-**

This selectors will filter objects based on set of values ex:- ‘env in (prod)’

* Here we use operators like in, notin, exists

1. Ingress:-

Ingress is an alternative to load balancer. In ingress we define routing rules that allow external users access cluster services

* Ingress allows external users access services in the cluster

There are 2 components in ingress

1. Ingress resource:-

In ingress resource we define routing rules like how request are routed to cluster services. We define in a yaml file

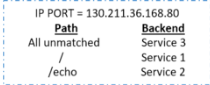
1. Ingress controller:-

Ingress controller is a daemon which acts as k8 pod. This ingress controller will route traffic to corresponding service as defined in ingress resource. It also tracks if any service is changed

* There are many ingress controllers like nginx controller, traefik, haproxy, gke, istio….

Types of ingress:-

1. simple fanout = path based routing

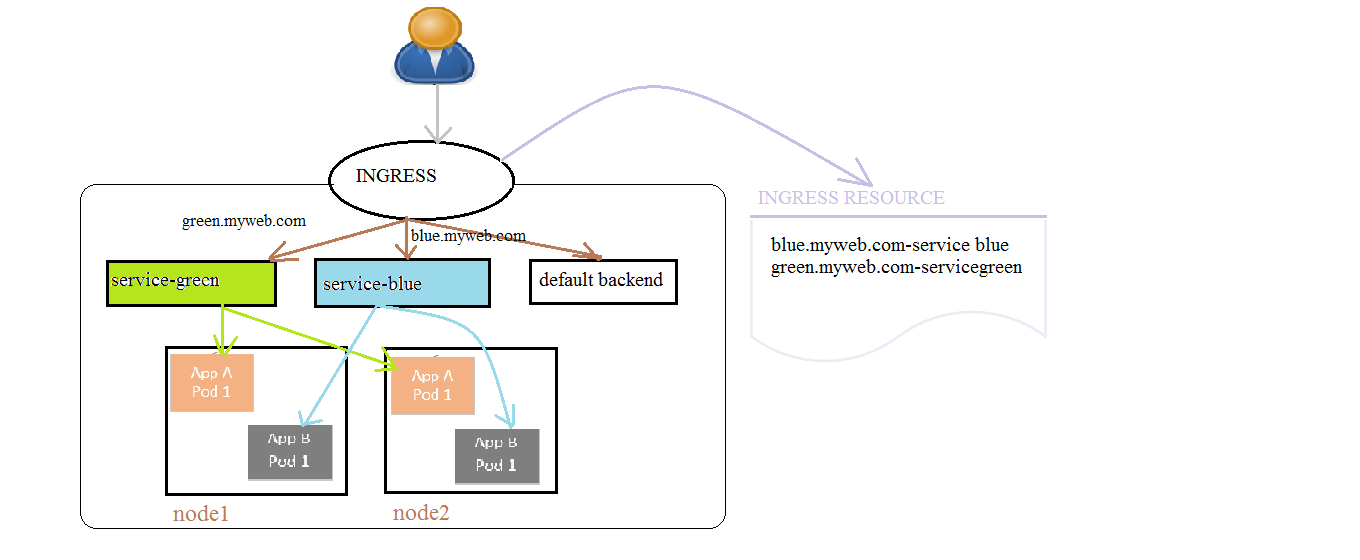


1. name based virtual hosting = it uses multiple hostnames for same ip address



\*default backend = an ingress with no rules. When a user access a service which is not defined in ingress resource then traffic will be routed to default backend

1. tls =
2. single service ingress = exposing single service through ingress

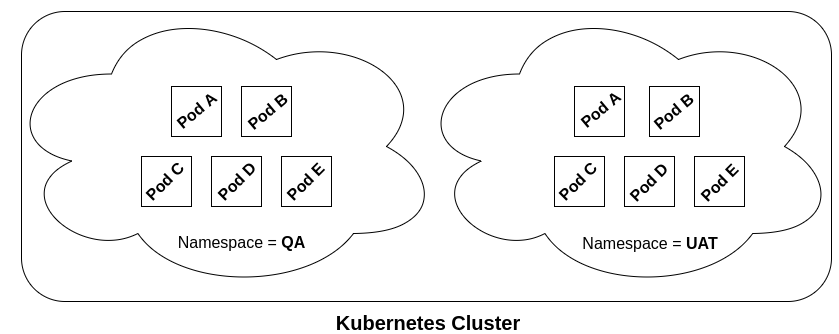


Assume there are 2 sites called blue.myweb.com and green.myweb.com.

Now user hits blue.myweb.com then this request will first hit to ingress endpoint either with http/https. From there the request will go to blue services and blue service will route to respective pod

The above process is one of the type of ingress which is name based virtual hosting ingress rule

1. Namespace:-



On a physical k8 cluster we run production environment, what if we want one more cluster for staging or testing environment. We can’t

Setup new clusters which incur lot of money and complex management. Also there will be a chance of name collision like creating same service names and deployment names ex: - I have k8 cluster where multiple teams deployed word press application. Then it would be difficult to identify which word press application is whose? And also we will get name conflict

So we create virtual clusters on top of physical clusters. These virtual cluster are called as namespace. These virtual clusters are isolated from other. You can use one name space for environment (or) for user (or) for application

* by default there are 3 namespaces created

(I) default namespace---by default all resources are created in default namespace in k8 cluster

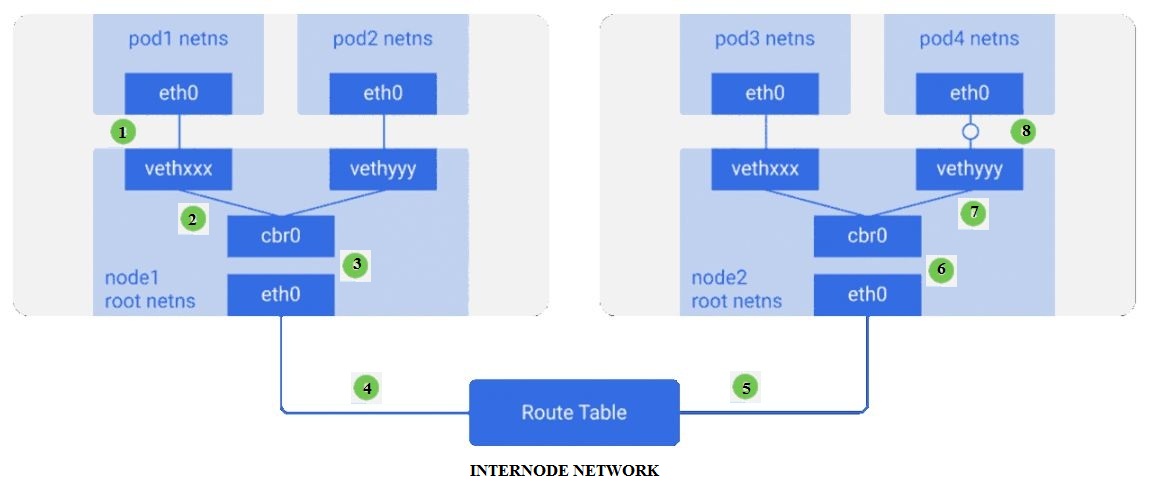
(II) kube-system-------namespace for resources created by k8 system

(III) kube-public-------namespace for resource that are publicly available

* We can’t create namespace for few resources like volumes...

1. Networking:-

Docker networking model is limited to host. So we go for kubernetes networking model which uses cni (container network interface) driver. We have many 3rd party networking plugins like flannel, calico, romana & weave net…..



1. eth0 = pod network namesapce
2. vethxxx = virtual Ethernet cards
3. cbr0 = linux bridge, if all container want to talk to other container it is possible through linux bridge with in the host
4. eth0 = root namespace (host network interface)
5. rout table = which is used to route the traffic from source to destination

Assume a packet is travelling from pod1 to pod4 then

1. Packet leaves pod1 network namespace at “eth0”and enters root namespace at “vethxxx”.
2. Now the packet moves to “cbr0”. This “cbr0” routes the packet to destination using ARP.
3. Now the packet comes to main network interface of the host “eth0” and finally goes to route table
4. This route table will send to the destination node’s main network interface of the host “eth0”
5. Now the packet will be forwarded to “cbr0” and sends it to “vethyyy” of root network namespace
6. Finally the packet reaches to pod4 network namespace at “eth0”

* Container to container communication is through local host and ports
* Pod to pod communication is through cni like flannel, calico
* Pod to service communication is through nodeport, dns
* Service to external client is through load-balancer, nodeport, ingress

1. Replication controller:-

Replica = copy of pod

A replication controller ensures the desired number of pods are always up and running at any given point of time.

Ex: - Assume you defined three replicas then 3 pods will be created. If one pod got deleted/terminated/failed/crashed then replication controller immediately creates a new pod. If there are more pods running then replication controller deletes the excessive pods

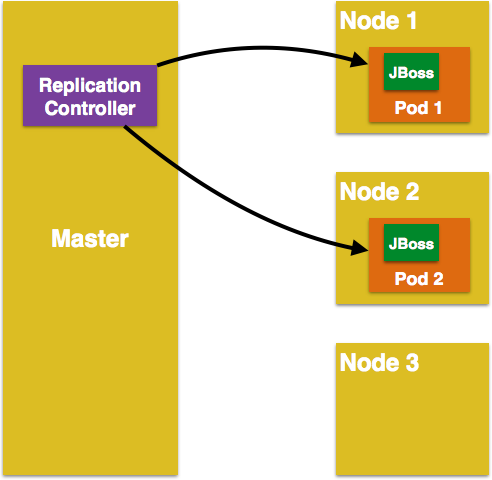
* Replication controller have pod templates. Using this templates RC will create any number of pod copies

\*Assume if renamed a pod how then RC keeps track? That is by using selector and label (equality-based selector ex:- env=prod)

Why replication controller:-

A pod started manually is of no use. Since pod restarts if it crashes. So we go for replication controller

1. Reliability = if one pod dies, RC will create a copy of same pod
2. Load balancing = load balances traffic to multiple pods
3. Scaling = RC will scale up & down when load becomes too much on the existing pods



1. Replica sets:-

Replica sets are new improved version of replication controller.

* Earlier we can use replicas either in replication controller or deployment kind. With replica sets we can use only in deployment kind
* Deployment can manage(take care of) replica sets

In replica set we use match labels instead of labels which is called as set based selector (environment in (dev, qa))

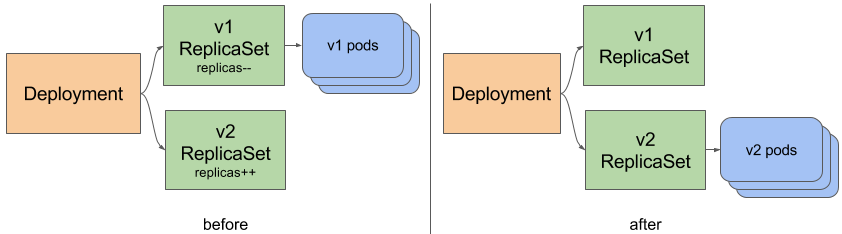
1. Deployments:-

Deployment is a controller for pods and replica set to control on how and when a new pod version should roll out and roll back

Why deployment:-

When you are ready to update the application. You need to introduce another replication controller. Which is bit difficult to manage. So deployment object will come into picture.

Deployment controller will monitor, upgrade, downgrade & scaling pods without downtime by maintaining multiple replica sets we describe desired state then deployment controller changes the actual state to desired state at a controlled rate

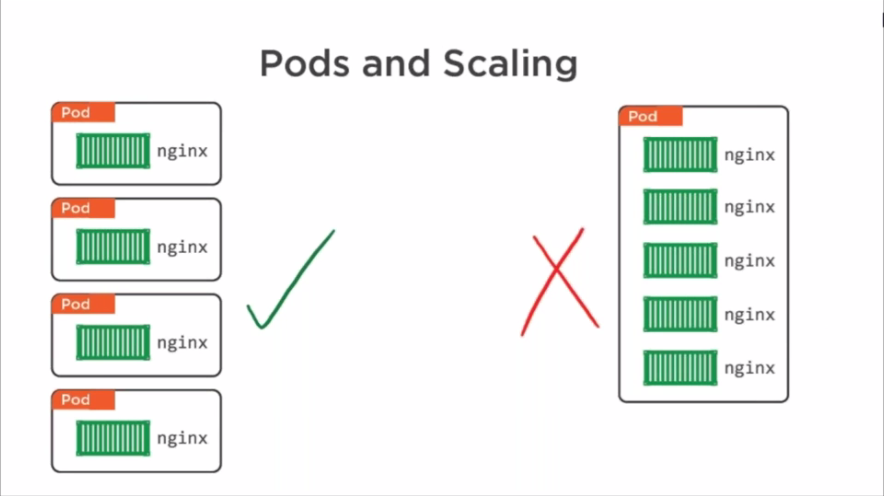


Deployment use case:-

1. create = deploying an application
2. roll back = roll back to previous version
3. update = deploying a new version of application
4. delete = pause/cancel the deployment

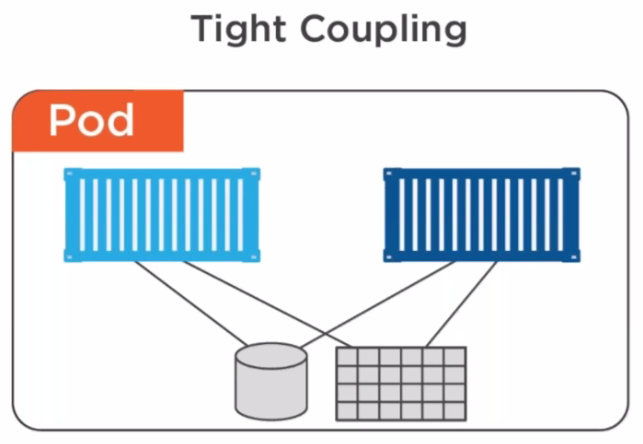
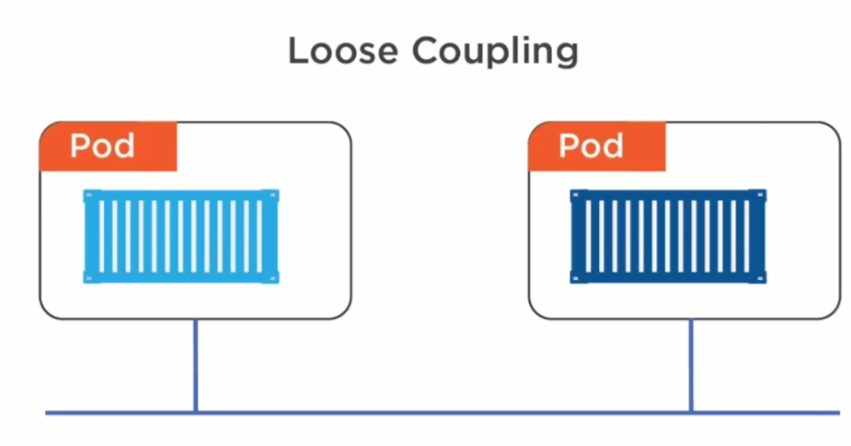
Deployment strategy:-

1. Recreate = will kill old rc and bring up new rc. However we have downtime
2. rolling update(default)
3. blue-green = kills one old pod and bring up one new pod and then switch traffic to new pods
4. canary = bring up few pods and if that is working correctly then proceed full and switch traffic to new one and finally delete old one



Whenever we want to scale the pods they must be loosely coupled

Loosely coupled means one container running in one pod where as tightly coupled means many containers running in one pod

1. Secrets:-

Secrets are Kubernetes objects. Secrets is any sensitive data like password, ssh keys, authentication tokens, ssl certificates. So putting sensitive data in pod definition or docker image is not safer. So we either put secret data in yaml file or plain text. The sensitive data will be injected into pods at runtime

\* all your secrets will reside in etcd and in tmpfs

1. Config maps:-

Application needs configuration. Configurations are config files (ini, json, xml...) or environment variables. These are stored outside the application because they change from environment to environment (staging, production, testing...)

* In container environment

1. Assume you have embedded configuration in an image. Now if your configuration is changed you need to rebuild the image.

2. Docker lets you specify env variables in docker file. Now you build an image and run a container out of it. Your app uses env variables, but what if multiple containers wants to access same env variables, it is not possible

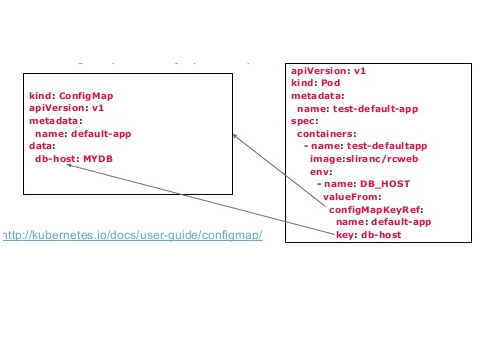
* so kubernetes confg maps comes into picture

config maps are kubernetes objects. config maps holds configuration in centralized place that are accessible to all containers. config maps are simple key value store

There are 2 ways to provide configuration to your application (i) by injecting env-variables with config maps (ii) config maps as configuration file

* whenever you change configuration just restart the pods to reflect the changes
* first create config maps then create pod and enter config maps details

\*config maps and pods must be in same namespace to work



1. Daemon sets:-

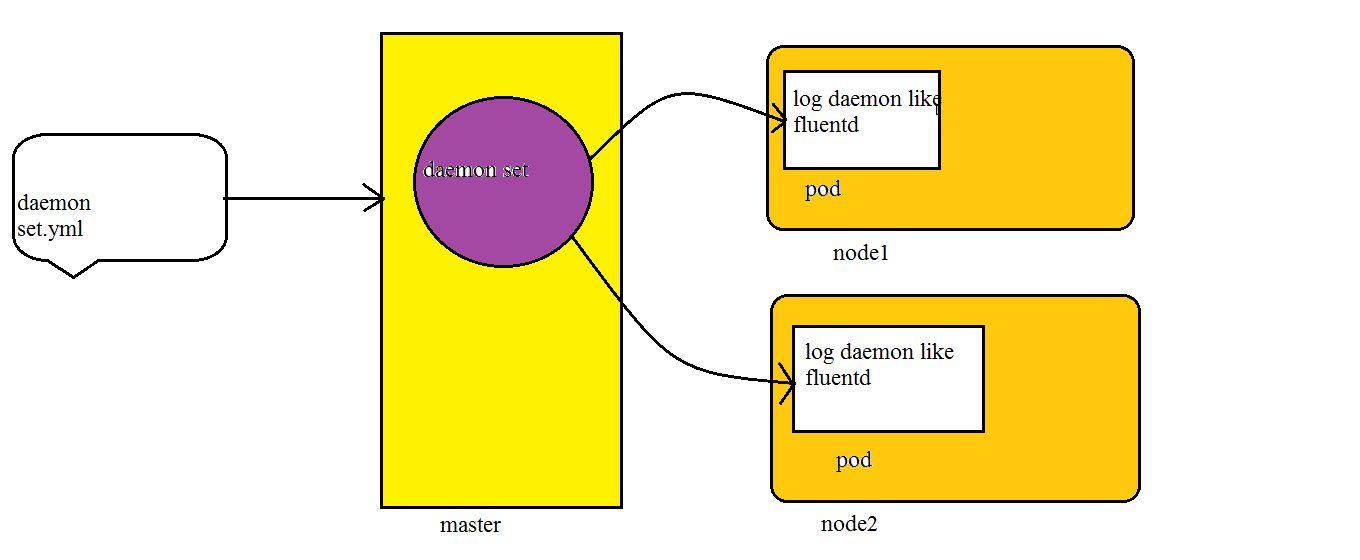
Sometimes we want to run same pod on all nodes for logging, storage, monitoring

Daemon set ensures that each copy of pod is running on all the nodes in the cluster. Those pods run as daemon. If you add a new node in k8 cluster then daemon set will automatically create a pod in that node. If you remove a node then daemon set will not reschedule pod on other nodes.

If you delete daemon set then all pods created by daemon set will be removed.

We go for daemon sets only when we wanted to run daemons in each and every node. Daemons like

1. Storage daemons like ceph, glusterd… to be deployed on each node for persistence storage
2. Logging daemon fluentd, logstash… to run on every node to collect logs of k8 components
3. Monitoring daemon collectd, gmond…. To run every node to monitor containers.
4. Ingress controller pod to run on every node to update ingress resource



Here I have defined in my manifest file to run logging daemon. Since there are 2 nodes daemon set created 1 pod in node1 and other pod in node2. If I add one more node then daemon set will create one more pod in new node.

\* We can’t create auto scaling for daemon sets

\* Alternative to daemon sets are init scripts

Node affinity = allowing pod to run on particular node

1. Resource quota:-

When multiple users/teams share same k8 cluster, there is a problem where one team can use more resource like cpu, mem than required and other teams may have shortage of resources. We overcome this problem with resource quota

* So we create namespaces for each team/user and enable resource quota on it. one resource quota per namespace
* admins create namespace and enable resource quota whereas developers will control the amount of cpu, mem for container by setting resource requests and limits

There are 3 types of resource quota

1. compute resource = cpu, mem
2. storage resource = pvc
3. Object count = pods, service, secrets, config maps, replication controller.

So we can impose two types of constraints on compute resource.

1. Request:-

Asking how much resource you want to use/the minimum amount of resources the containers needs/requesting the amount of resource for

Container/requests are what containers guaranteed to get/assigning this much values to containers

Ex: - there is node1 which has 1 GB resource and node2 with 3 GB resource. Now you asked 2 GB for container, then scheduler will search both the nodes and see where 2 GB resource is available and then assigns pod on to that node. Scheduler will not overcommit request (if you ask for 2 GB then scheduler will not give 3 GB resource)

1. Limit:-

The maximum amount of resource that a container can use. The container cannot utilize more resource than specified.

Ex: - if I give 40mib as memory limit, then container cannot exceed 40mb of memory

* cpu is measured in cores or millicpu where 1 core = 1000millicores. If your containers need 1/4th cpu then mention cpu request = 250ms memory is measured in mib/gib/tib...

\* If we create resources (pods, services...) more than the resource quota then we will get a 403 FORBIDDEN error ex: - if a pod requests 3 GB but resource quota is set to 2 GB then we will get error

* The admin can also set object limits like 10 config maps/namespace, 15 pods/namespace, 5 services/namespace....
* don’t put any restrictions on production
* resource quotas are only for containers if pod have multiple containers then we can set requests and limits for each container

In resource quota you can restrict total usage of cpu and memory per namespace like



1. requests.cpu:- the sum of all containers cpu request per namespace

Ex:-I have a namespace "dev." where I defined requests.cpu as 500m means you can have 1 container with 500m cpu request or 50 container with 10m cpu request or 5 containers with 100m cpu request

1. requests.memory:- the sum of all containers memory request per namespace ex:- if I give request.memory as 100mb, then you can have 50 containers with 2MiB requests or 5 containers with 20MiB CPU requests or 1 container with 100mb request
2. limits.cpu:- the maximum combined cpu limits for all the containers in the namespace
3. limits.memory:- the maximum combined memory limits for all the containers in the namespace
4. Health checks:-

Your pod and container can still run if your application is not working. To detect such problems we go for health checks.

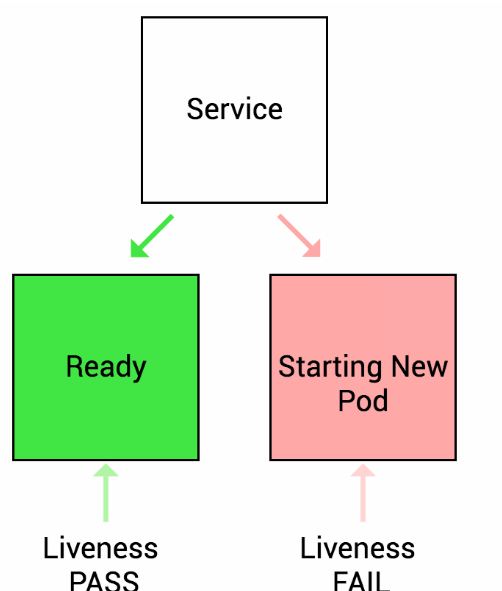
* Health checks can determine if your application is running or not. if the instance of app(application in container) is not working then service will not send request to that container
* by default k8 will send traffic to pod when all containers are started and restarts when a container is crashed

Types Of Health checks:-

Liveness and Readiness probes really help with the stability of applications.

1. liveness probe:-

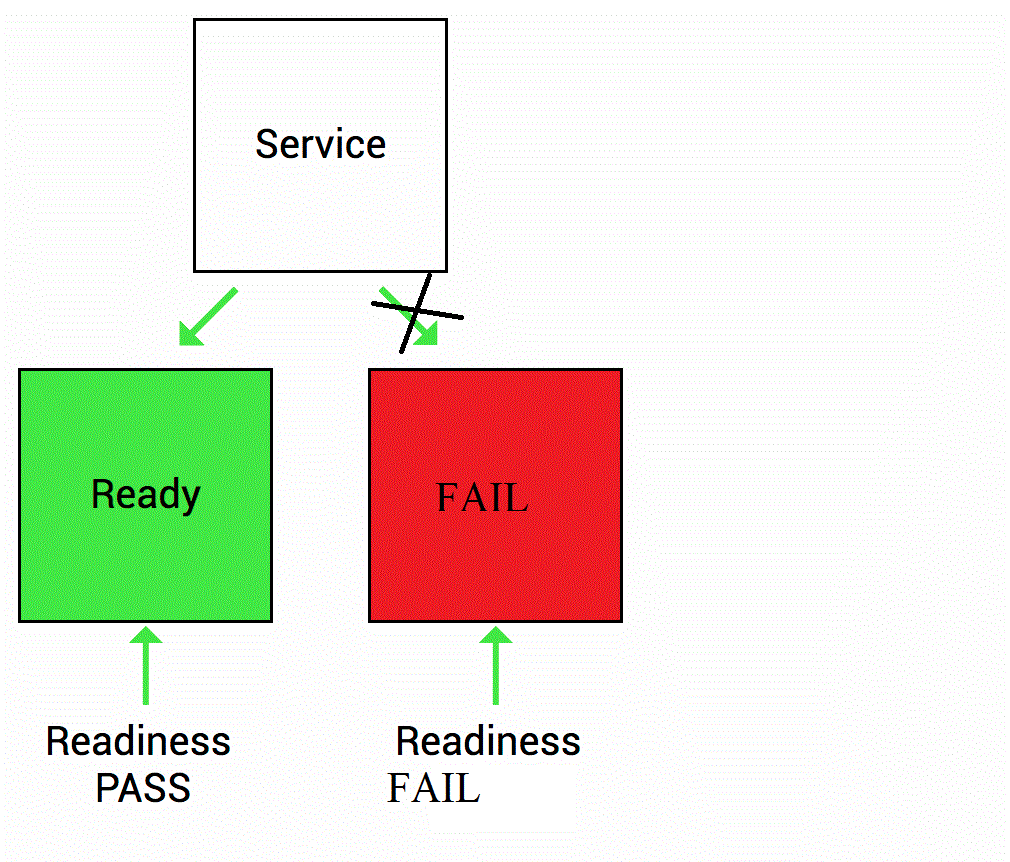
With liveness probe kubernetes can know if your app is alive or dead. If app is dead then kubernetes removes that pod and replace with new pod



1. readiness probe:-

With readiness probe kubernetes can know if your app is ready to serve traffic. If that app fails then that pod is removed from

Service load balancer



Types of Probes:-

There are 3 types of probes (probes = testing readiness/liveness)

1. http = 200-300 as output then probe is success

2. Command = $?=0 then probe is success

3. tcp = if establish tcp connection on a port then it is success

Probe Options:-

1. Initial delay seconds = 15, after container started it is waiting 15 sec to start probe

2. Timeout seconds = 2, after 2 seconds if there is no response then it mark as failure

3. Path = /index.html, kubernetes pings this path and gets a response then marks app as healthy

4. Period seconds = 10, for every 10 seconds it will do probing

5. Success threshold = 4, after probe failed, if probe consuctvly passes then it marks as success threshold

6. Failure threshold = 3, if probe fails 3 times then it will be marked as failure threshold

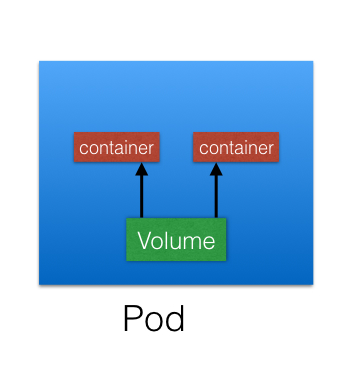
.

1. Volumes:-

We know that pods are ephermal (live for short period), so the data in the container will also be lost. We want the data to be persistent.

1. Imagine you spin up a container from an image, now all the data is stored in container, if container dies. You will spin up one more container from the same image but you will not find any data
2. We can’t share data between multiple containers
3. you cannot attach a physical disk to logical container to store the data, but k8 allows this functionality

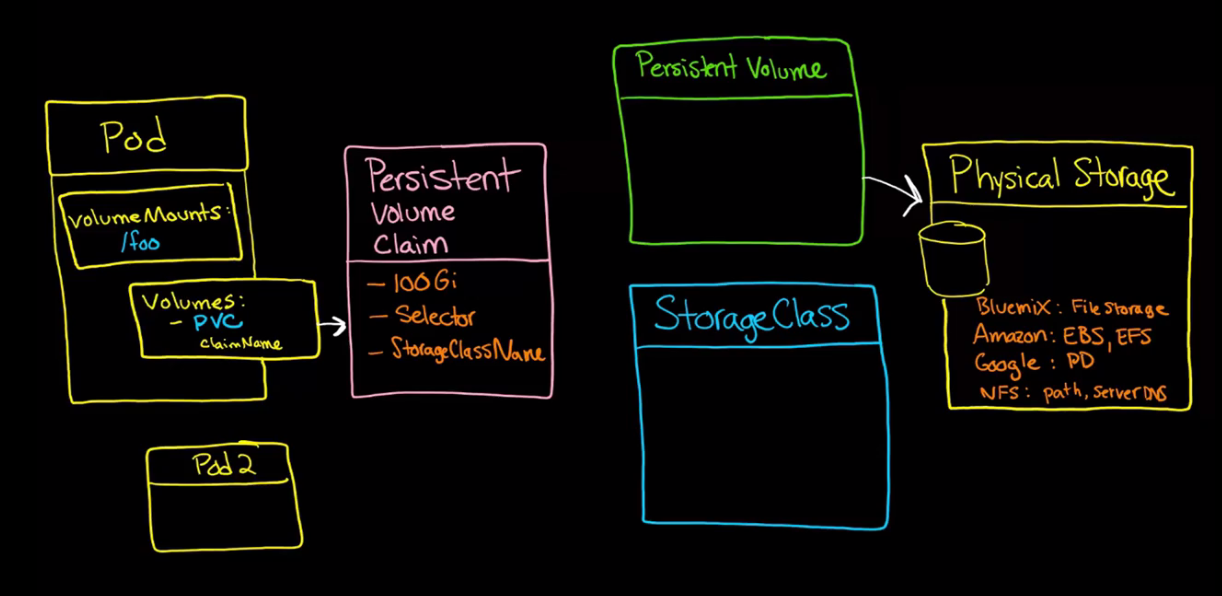
* So a k8 volume is a special directory, which is backed by storage medium

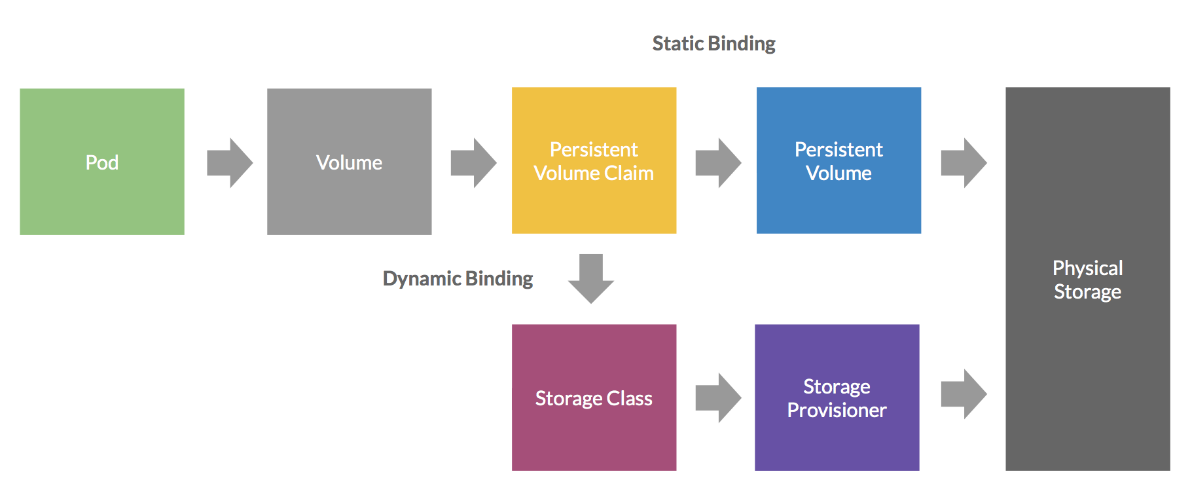


TYPES OF VOLUMES:-

1. host based = emptydir, hostpath = these volumes exists in host/node, if the node goes down we lose the content
2. cloud provider = aws ebs, azure disk, gce persistent disk, vsphere volume
3. distributed fileystem = ceph, aws efs, aure filesystem, gluster = file sharing type volume
4. Others = iscsi, fc, vsphere, flocker...
5. file sharing = nfs
6. special purpose = secret, git repo, config maps

There is a pod in which has volume mounts (volume inside a container) “foo”. Now we will reference pvc details in pod volumes. Now the pod is attached to pvc and pvc will bind to one of the pv



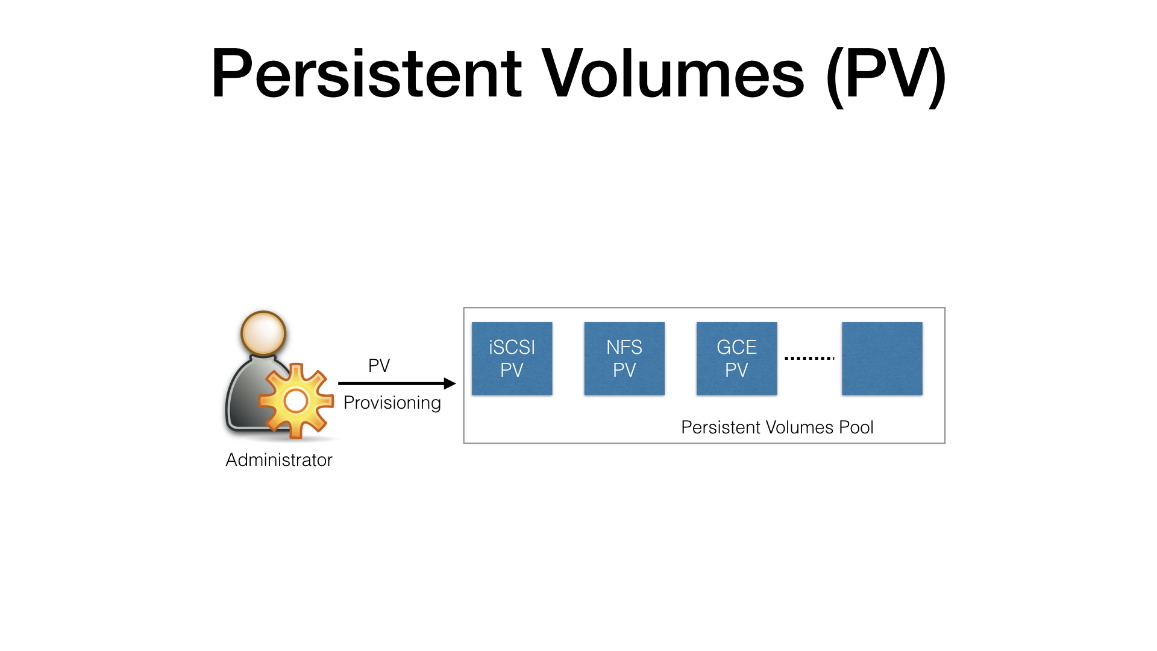


So the storage is divided into 2 types

(I) Persistent volume (pv):-

pv is like a volume plugin which uses underlying physical storage. These pv's are created by kubernetes admins with particular file system, volume ID, size, reclaim policy. These pv's are backed by physical storage (either ebs, gce,azuredisk, nfs).

The admins can create a pool with one or more pvs we define all the volume details in yaml file.



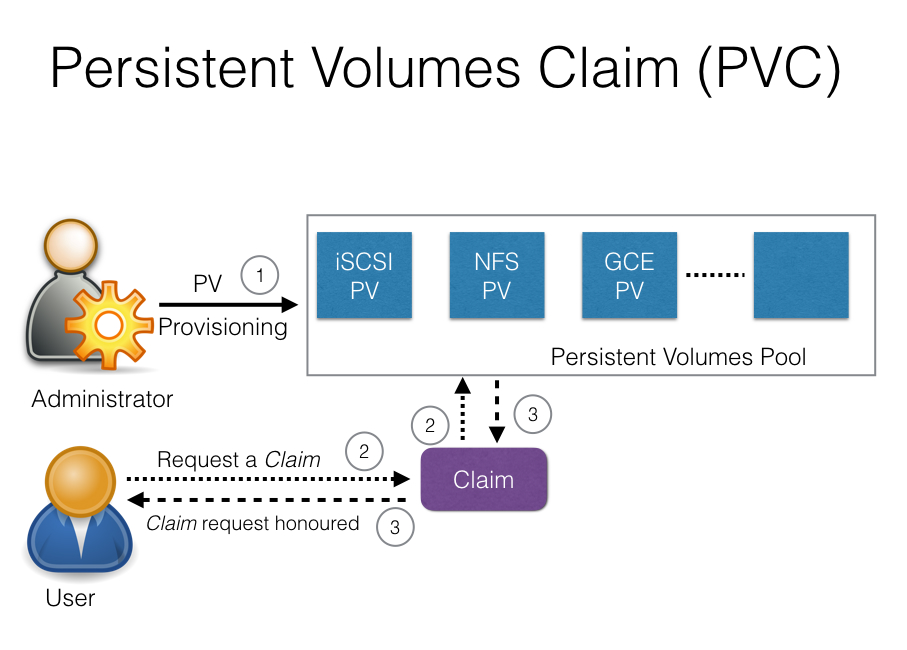
Developer/user who is working on volumes in pod are not aware low level details of storage like file system type, volume-id. So comes pvc

(ii) Persistent volume claim (pvc):-

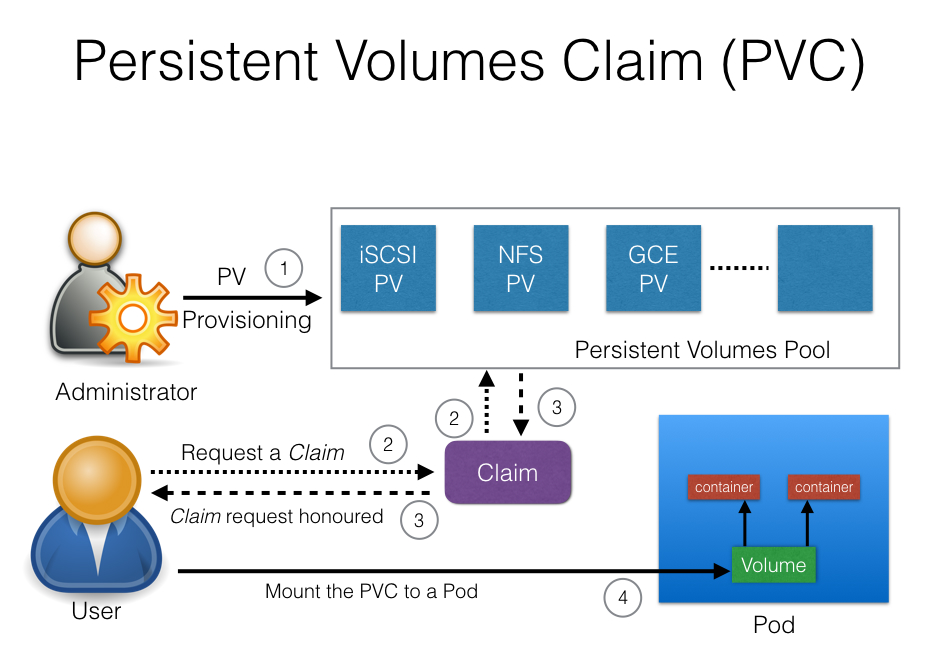
A user requests for storage which is called as pvc. Pod can’t access volumes directly. So pod has to access pv through pvc, in pvc we define desired size, acess type, selectors, storage class name. Developers create pvc in yaml file and refrence/mention in pod

\* You cannot bind two pvs to same pv but you can use same pvc in two different pods

\*If there are multiple pv's then how will pvc select a pv. By using selector



Kubernetes cluster admin will create pv. Developer/user will create pvc. Now we will reference pvc details in pod. Now pvc search in the existing pool of pvs and attach best possible match



There are 2 types of volume provisioning

1. Static Volume Provisioning:-

Before user deploy application, user willa ask cluster admin to create pv of your desired size. Once pv is provisioned then user will create pvc.

* Static volume provisioning is pvc cannot be claimed unless pv is created.
* Static provisioning refers to persistent volume
* Assume you mentioned that you want 10 GB storage in pvc, but a pv with 12 GB got attached in which 2 GB got wasted.

1. Dynamic Volume Provisioning:-

When you create pvc automatically pv is created

1. Storage class:-

Dividing the storage infrastructure based upon speed, type of storage…

And each class is called storage class.

Ex:- fast class, slow class, gold class, platinum class, …..

* Storage class is one time setup created and configured by admin
* Admin has to create default storage class because when developers failed to mention storage class in pvc, then while pvc is created automatically it is attached to default storage class

Here cluster admin will create storage class, you create pvc and mention storage class. Whenever pvc requests automatically pv will be created and attached to it

* Dynamic volume supports by few storages like gce, azure, aws, ebs....
* Dynamic provisioning refers to storage class. Binding is attaching pvc to pv exclusively such that two pvc cannot bind to one pv

Reclaiming:-

1. Reclaim policy will tell you what to do after the volume is released from claim means whether to delete, recycle, retained
2. Assume a pv is bonded to pvc. Now you no more want pv. so now what shall we do with this pv whether to delete/recycle/retain the volume

Reclaim Policies:-

What happens when you delete a pod/deployment that uses pv

1. Retain = this is default. Once the pod is deleted still pv exists, your data will remain. Pv will be there until your cluster admins delete.
2. Recycle = deletes the content on volume and make its available for new claim. Dynamic as well depreciated.
3. Delete = when you delete pod and pvc, automatically pv is deleted too

Access modes:-

Assume there is a deployment which has 2 replicas out of which one pod is running on one node and other pod on other node then how to access pv and which one to access

1. readwriteonce(RWO) = the first pod on first node will be able to read write to pv and other pods will be able to read pv
2. readwritemany(RWM) = the first pod on first node as well as second pod on second node will be able to read write to pv
3. readonlymany(ROM) = the volume can be mounted as read only
4. Auto scaling:-

There are 3 types of auto scaling in kubernetes

1. Horizontal pod auto scaling = increasing/decreasing number of pods
2. Vertical pod auto scaling = scaling the resource of a pod like cpu, mem….
3. Cluster auto scaling = pods need to be scheduled on nodes when ther is no enough space in node then it will scale nodes

Horizontal pod auto scaling:-

HPA is a k8 resource/object. Horizontal pod auto scaling allows Kubernetes to scale number of pods in deployment/replication controller/replica set based on the metrics. Ex: - whenever average cpu utilization of pods cross the threshold value that you defined in manifest files, then auto-scaling controller will scale the pods. The average cpu utilization is collected by monitoring daemon which is deployed in a pod. This auto-scaling controller will check continuously

Types of metrics:-

1. cpu utilization = if cpu metric cross threshold value then k8 scale the pods
2. memory utilization = if memory metric cross threshold value then k8 scale the pods
3. transactions-per-second utilization= if tps metric cross threshold value then horizontal pod auto scaling happens

Configuration options:-

1. spec.scaleTargetRef.name = wheather to scale deployment/replication controler/ replica set
2. spec.minReplicas = minimum number of replicas
3. spec.maxReplicas = maximum number of replicas
4. spec.targetCPUUtilizationPercentage = threshold value

.

1. Jobs:-

Job is a controller in Kubernetes. The purpose of job is to create one or more pod and track about success of pod. We have 2 types of jobs

1. Job(run to completion):-

We submit the job manifest file to api server, the pod will kick in and execute a new task. When the task is successfully completed then pod goes to stopped state this is called run to completion. Job controller will wait for exit code “0” then pod will move from running state to shutdown state. We have to manually delete these pods. If one pod crashed while executing task then job controller will launch a new pod and continues the task from middle

* There are 2 types of jobs

1. Non-Parallel jobs = one pod is started and terminates if task is completed successfully
2. Parallel jobs = running one or more pods at same time
3. Cron job(scheduled job):-

Similar to Linux cron job

1. Monitoring:-

Monitoring the entire k8 cluster without downtime is more important

There are 2 types of monitoring

1. cluster monitoring:- here we monitor how many nodes are running, how many applications are running on each node, resource utilization like cpu, memory, bandwidth
2. pod monitoring:- here we need to monitor pod’s health

Monitoring tools:-

1. heapster
2. Prometheus\*
3. Grafana
4. CAdvisor
5. Datadog
6. influxdb
7. Helm:-

Helm is a Kubernetes package manager just like rpm/yum in redhat. Till now we have written our own yaml files for replication controller, services, deployment but helm comes with charts. These charts are pre written manifest yaml files

\*monocular is gui of helm

1. Node:-

Node controller is one of the control manager which is responsible for maintaining node objects.

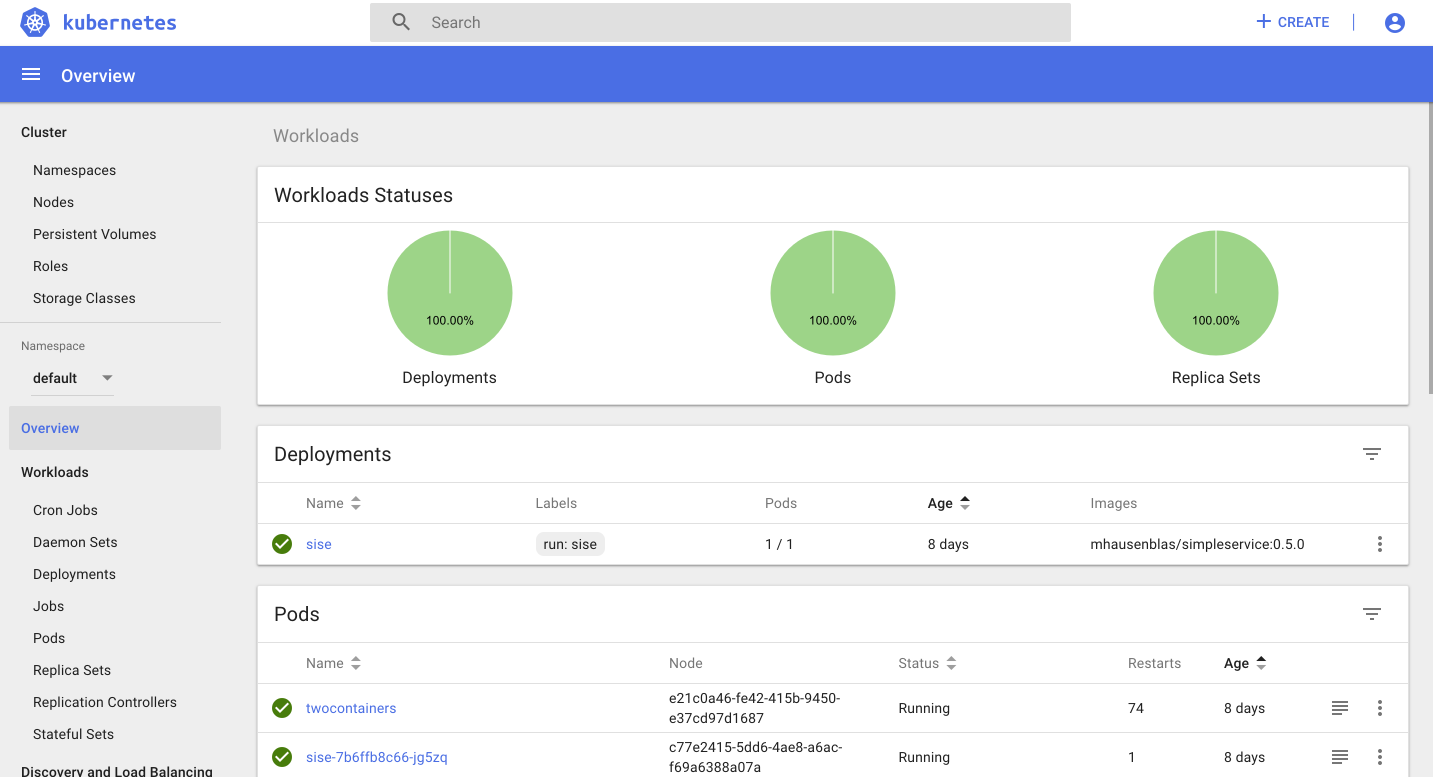
* Node controller assigns ip space to newly launch.
* Node controller keeps node up-to-date
* Node controller also monitors health of node
* A new node object is automatically created with metadata and labels
* A node has node condition like ready, out of disk

You decommission a node so you do t gracefully by moving all the pods to other nodes. So you drain a node before shutting the node

1. Dashboard:-

Dashboard is used to give you graphical overview of how many pods, services, deployments, configmaps, secrets, namespaces, volumes running,

* In dashboard we can troubleshoot the pods, deployments, secrets.



1. Logs:-

There are 2 types of logs

1. Application log:-

Pods generate application log

1. Cluster log:- Api server, scheduler, kubelet, node generates cluster log \*you can use either fluentd or elk for logging