Kubernetes

Author : Ankit kumar Panda
Date : 16 May 2020
GitHub: akpanda.ds@github.com

Introduction

Containerization vs **Virtual Machines**

Containerization is a technology which allows all the application dependencies and libraries to be packaged under one image. This enables development of a single file deployment.
Containers images run in islolation from one another, but run directly over the kernel.
The host machine resources are shared among all containers but sufficient isolation is provided to ensure that the images do not use resoueces overlappingly.
The isloation of conatiner images is acieved by 2 methods -> 1. Namespaces and 2. cgroups

Namespaces provide processes with their own view of the system. The process has no visibility to the world outside of its own namesapce.This enables isolation of the containers multiple namespaces contain pid,net, mnt etc

cgroups or *control groups* involve resource metering or monitoring. It limits what one process can use. cgroups involve memory, CPU, block I/O, and network

One important things to note is that container images are immutable, Once created they cannot be changed. Any change in the container image will result in a new image being created.
Since the container images conatin only the dependencies of the application, anduse the host machine OS and kerner directly , these images are light weight.

There are a number of container technology providers, for eg podman, docker and rkt.

Virtual Machines have logical seperation and run on the hypervisor.

Each VM comes with its own host OS. Starting and stopping of VMs take more time as there is starting of OS.
Each process running on the VM accesses the kernel of the VM OS rather than the actual host.

Each VM appears as if tunning its own bare hardware, giving the appearance of multiple systems on the same actual hardware.
Multiple applications have can run in an virtual machine

Kubernetes and Docker

Kubernetes
- is an container orchestrator engine, which can be used to manage deployments.
- comes closely integrated with cloud technologies but can also run on baremetal technologies.
- became popular because it has the ability to convert isloated containers running on different hardware and group them into a cluster
- is supported by major cloud providers.

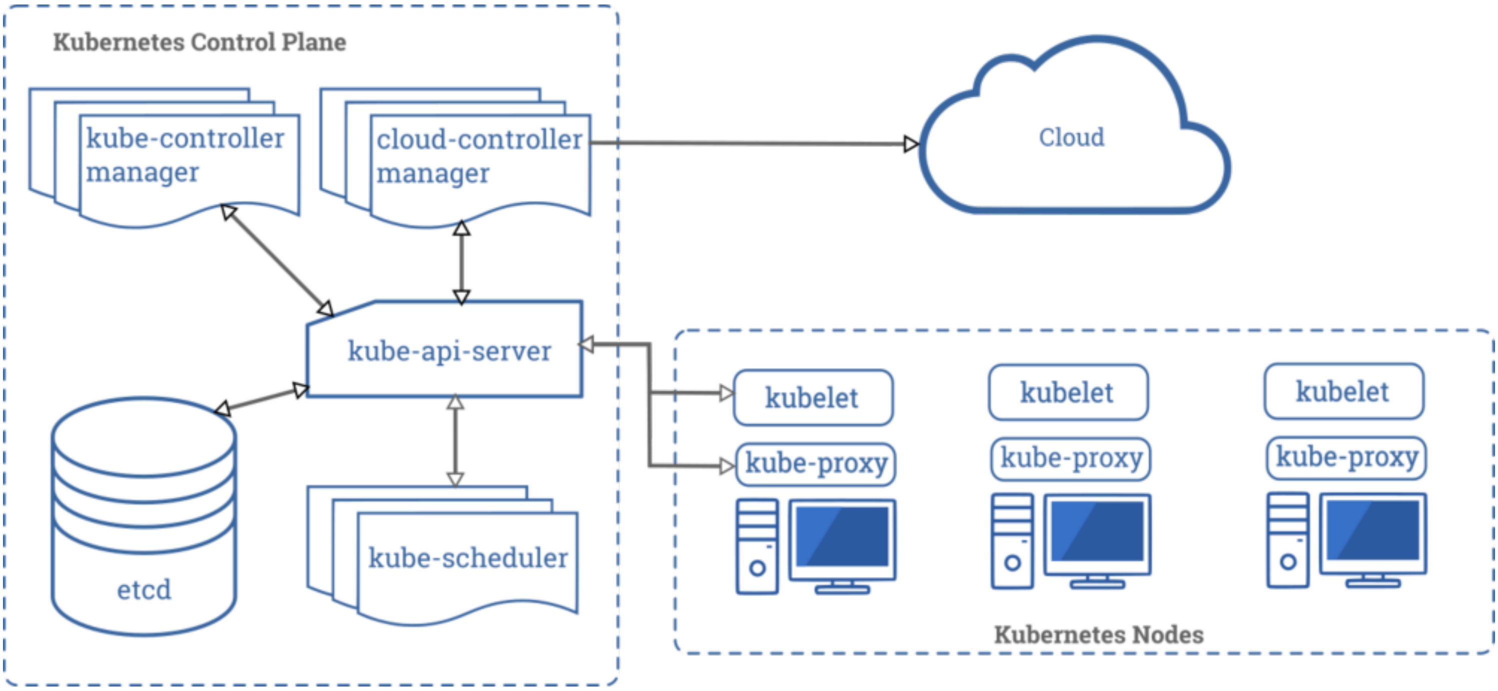
- provide autoscaling and auto healing and routing properties

Docker images run on a <u>docker container engine</u>. This can be compared to jars running on a JVM

Kubernetes Ecosystem

The kubernetes software spins up different nodes.
One or more of those nodes are designated as the master node. The concept of having more than one node is to ensure high availability clusters,
The master node is also known as the Kubernetes Control Plane.
Several kubernetes processes run on the master node to ensure kubernetes is running smoothly.

The different components can be visualized as below (taken from the kubernetes documents).



<u>Components of the Kubernetes Control Plane</u>:

1) *kube-api-server* : The frontend of the master node. All requests to the control plane pass trhough the api server. Internal implementation of requests is through RESTFul calls. hit by User requests with json or yaml

2) *etcd*: A distributed key value data store. Cluster store for stroing all the meta data with respect to cluster information an node. Cluster store is built on etcd and serves as the single source of truth.

3) *kube-scheduler* : This takes care of the pods creation taking into account affinity, locality data strore etc. when new pod requests come in.

4) *controller manager* : Could be cloud-controller-manager if kubernetes is running on cloud. If running on bare meta machine then kube-controller-controller. Responsibility is to

maintain the actual state of the cluster and node as per the expetcted state.

1- Controller manager runs a numbe rof DEAMON processes (infinite loop processes) to ensure the states remain consistent with the extpected states of cluster
2- Has different sub-componenet for node controller, replication controller, router and volume controller manager

<u>Components of a Node</u>:

1) *kubelet :*
   1- Agent running on each node. Part of kubernetes software.
   2- Takes order from master node, gives status of the order and conveys messages to master node.
   3- The agent is exposed on each node on 10255 port
   4- Any linux machine when is installed with with kubelet acts as a kubernetes node and tries to find and listen to the master node

2) *kube-proxy :* Ensures that each pod has an IP address when bought up. IP addresses are ephemeral. Works closely woth Service Object

3) *container-engine/runtime :*

   1- CRI (Container Runtime Interface) interacts with kubelet
   2- pulls images from the repository
   3- starts and stops containers
   4- Currently docker and rkt is supported by kubernetes

Common Conecpts

The basic componenet of kubernetes that can be deployed is Pod,
Kubernetes cannot communicate directly with container but it occurs through Pods.
Pods contain one or more containers(usually one) and they provide isolation from other pods through a sandbox approach.
Pods are *<u>atomic</u>* in nature, meaning either all conntainers of the pod are deployed or none are deployed. Pods cannot span across multiple nodes.
Pods can be created by direct pod creation *command, replication controller or deployements.*
Pods have ip addresses but bith the pod and the ip address asoociated with it are *ephemereal.*
This gives rise to following higher level kuberntets objects:

1) Replication Controllers to maintain and heal pods
2) Deployment to maintain versioning of pods
3) Service to give a static or non changing address to pods or a service

<u>Master Node Communication</u>:

From cluster to master: happens at api-server only. https with client aithentications. nodes must be provisioned with appropriate public root certificate. can take place iver public network as the communication is secure

From master to cluster:
1. From apir server to kubelete : Not safe be default. Not run on public networks. To add

escurity
set kubelet-certificate authority amd use ssh tunnellinh
2. api server to nodes : uses http. Neither authenticated nor encyrpted.

<u>Where to Run kubernetes</u>:

1) Public cloud : AWS, Azure, GCP, etc -- support from providers
2) Bootstrap : On prem , private cloud --- kubeadm
3) Playgrounds pwk, minikube(not suitable for prod

<u>Communication with Kubernetes</u>:

- *kubectl* : most common.POST requests
- *kubead* : create cluster from different indiviudal nodes
- *kubefed* : Nodes in multiple clusters, Partly in public cloud or in multiple pubilic cloud.
- could be programmatic calls using client

Almost everything in Kubernetes is an Object. Object management can be done by

1) Imperative commands (kubectl run, expose etc)

2) Imperative object configuration (kubectl + yaml or config files)

3) Declarative (preferred. just tell what needs to be dona nd not how)
Most robust and most complicated.
kubectl apply -f config.yaml

<u>Declarative</u>:

Three things to keep in mind while using delcarative

1) Live Object configuration : live configs as observer bu the kube cluster

2) Current Object Configuration file : config file currently being applied

3) Last-applied object configuration file : the last configuration that was applied

<u>Primitive fields are replaced</u> with Current Configuration values
Map Fields are *<u>merged</u>* with old map
List fields are complex and beyond the scopr of this document.

Objects are persistent entities in kubernetes
information about this is maintained in etcs

Identified using names (client-given) and UIDs (System Generated)
Can be only one unique name at a time.

<u>Namespaces:</u>

Divides physiscal cluster ino multiple virtual cluster.
Objects need to be unique within namespace.
Namespace should not be used for versioning. Use labels instead
Three pre defined namespaces are present
→ default : if none specified

→ kube-system : for internal kubernetes objects
→ kube-public : auto readable all users, even those not authenticated can use resources in public namespace. Used for cluster

Objects without namespace inclide low level objects like nodes and persistence bolume

Labels:

key vakue pair, need not be unique, loose coupleing via selectors
can be added during or after creation of object
key must be unique at object level

Service:

ClusterIP is assigned when service is created. Independent of backend pod lifespan. static port can be assigned. Any pod can talk to clusterIP.

```
selector:
    key:value
    key:value
```

The above format can be used to select pods based on labels.

Annotations:

key-value pair, metadata
used for metadata required by object itself. Not used for identifications. Can be long.

Volumes

Pods are ephemaral and the data is lost as soon as the pod is shut down
Volumes let the pods wrote to a filesystem that exists as long as the pod exists.
Persistent volumes are long term storage n=and exist beyond containers,pods and nodes.

To use a volume:
1) Define vokume in pod spec
2) Have each container mount the volume.
3) Different path for different container

To login into a pod

kubectl exec -it <pod-name> -- /bin/bash

Types of volumes:
Apart from cloud service providers which provide their own storage solutions, there ar ea few volumes that are inside kubernetes. Below are the important ones

1) *configMpas*
   - key-value pairs, mounts data from ConfigMap Object, Inject parameter into pod
   - Usecases : provide config info for apps in pods, specify config info for control plane

2) *emptyDir*
   - not persistent, initially empty, shared across all containers in the pod

- created when pod is created, conatiners can mount differnet times
- Can survive container restarts within the same pod.
- Usecases: scratch space, checkpointing,

3) *gitRepo* (deprecated)
- It mounts an empty directory and clones a git repository into it for your Pod to use

4) *secret*
- pass sensitive info to pods, create secret first and stored in control plane.
- mount the secret as a voliume to be available inside pod.
- secret is stored in RAM storage and not on persistent volume

5) *hostPath*
- mount file from filesystem intp pod, results in pod-node tight coupling
- not common, pod should be independent of node.
- Usecases : Access docker internals, running cAdvisor, access block devices or sockets on host

Persistent volume are low level objects. 2 types of persistent volume provisioning

- Static: admin pre creates the volume
- Dynamic: Containers need to file a PersistentVolumeClaim

<u>Creating secrets</u>

From command line directly
kubectl create -f secrets-demo.yaml ----- the yaml file to contain key-value pairs
kubectl describe secret <secret-name>

From files
kubectl create secret generic sensitive --from-file=./username.txt --from-file=./password.txt

kubectl describe secret sensitive

After creating a secret mount it in the volume section of pod yaml to use it.

<u>Create configMaps</u>

kubectl create configmap fmap --from-file=file1.txt --from-file=file2.txt

or

kubectl create cm special-config --from-literal=special.how=very

Edit the env section (inside container level) in pod yaml to point to use these configMaps
Once edite emv section to add the confiig map, mount the volume in mount section

Pod and Containers

Before using a container image, it mist be used and present in a repository.
Repository could be private or public. If private authentication is required.
If using cloud techology, AWS Container Registry or Azure Container Registry etc could be used directly

If non-cloud registry, 3 ways of reading the repository is used
1) Configure nodes to authenticate to a provate repo (All pod can pull any image).
2) Pre-Pull images (Pods can only ise cachec imgae. Need root access during setup)
3) Use *ImagePullSecrets*. Pods withsecrets can pull image secrets. happens at pod level.

kubectl commands:

1) kubectl describe --- describe the status of the kubernetes object
2) kubectl edit --- edit the pod/object in live condition
3) kubectl apply --- declaratively apply a config file. could be created or merged. Any ops can take place
4) kubectl create --- imperative way of creating objects from yaml file

What env can containers see?

1) FIlesystem : image at root, associated volumes
2) Container : Hostname becomes the name of the pod where the container is running
3) Pod : Pod name, user defined env variables (via downward api , a special volume)
4) Services : list of all services

To list all the env variables in a pod
- ssh into the pod : *kubectl exec -it <pod-name> -- \bin\bash*
- exuecute command : *printenv*

Pass information from Pod to Container

Environment variables are great when needed to pass external information to containers. But to pass kubernetes information or metadadta *downward api* is used.

```
   volumeMounts:
    - name: podinfo
      mountPath: /etc/podinfo
      readonly: false
   volume:
    - name: podinfo
      downwardAPI:                       // downward API to access different part of pod spec
       items:
        - path: "labels"
          fieldRef:
           fieldPath: metadata.label
        - path: "annotations"
           fieldRef:
            fieldPath: metadata.annotations



Refer the downward-api.yaml in the git repor for the full yaml
```

Lifecyle Hooks (Spcecific to container, not pod as a whole)

1) PostStart : Called immidetely after container creation. No parameter. No guarantee that it will be execurte before the entry point of container

2) PreStop : before container terminated. Blocking hook. Must complete before terminting container

Hook handlers : To habdle hooks

Exec : shell commands
HTTP : Execute http against specific endpoint of container

Delivery of hook is atleast once.

--- check lifecyle-demo.yaml file for details

Node assignment to Pods

Usually pod node assignment handled by kube-scheduler
Usecases for more control:
- specific hardware
- colocate pods if they communicate frewuently
- ensure high availability

This can be achieved by:

1) **Node Selector**:
   tag nodes with label, add *nodeSelector* to pod template, pod will reside only on nodes selected by node selector. Simple but crude and hard constraint

2) **Affinity and Anti-Affinity**:
   2.1 -> Node Affinity : Steer pod to node. taint for anti affinity
   2.2 -> Pod Affinity : Steer pod to or away from other pods (affinity and anti affinity).

```
kubectl label nodes gke-first-cluster-default-pool-89d9341f-t2zc disktype=ssd

kubectl get nodes --show-labels

---------- yaml file start ---------

apiVersion: v1
kind: Pod
metadata:
 name: affinity-demo
 labels:
  env: test
spec:
 containers:
 - name: nginx
   image: nginx
   imagePullPolicy: IfNotPresent
 nodeSelector:
  diskType: ssd

------------- yaml file end ---------
```

Taint and toleration

Certain pods should never go to certain noodes. So mark or *taint* all of those nodes with a key-value
Taint is the anti affinity.
Only the pods which have tolerance for tha taint will be allowed on the tainted nodes

Taints can be added based on node condition by noce controller autmoatically

```
kubectl taint node gke-first-cluster-default-pool-89d9341f-t2zc env=dev:NoSchedule

---- the key value is env and dev. Effect is no pod scheduled on this node till they have tolerance
for the same


apiVersion: v1
kind: Pod
metadata:
 labels:
  name: toleration
spec:
 containers:
 - name: nginx
   image: nginx
 tolerations:  ------------------- this indicates the pod is ok to be scheduled with the below taints
on the node
 - key: "dev"
   operator: "Equal"
   value: "env"
   effect: "NoSchedule"
```

Use cases:

Dedicated nodes for certain users
- taint subset of nodes
- only add toleraions to pods of that user

Nodes with special hardware
- Taint nodes with GPU

Init Containers

- Run before app containers are started
- Always run-to-completion
- in case of multiple init container, all of them run serially. In case of failure, kubernetes restarts the pod till the init container runs

Usecases:
- Run Utilities that should run before app container
- Different namespace/isolation from app container
- Include utilities or setup
- Block or delay start of app container

Pod Lifycycle:

A pod can be in one of the follwoing states

1. *Pending* : Request recieved by master, but pod is not ready to serve
2. *Running* : Pod is ready to serve requests
3. *Succeded* : If all the containers are terminated succesfully
4. *Failed* : If one or more containers encountered any error while terminating
5. *Unknown* : the status of pod cannot be obtained

Probes

This is a diagnostic action performed by the kubelet periodically. This is performed by one of the handles implemented by the container. The probe ensure that the latest status of the pods is always known

There are 3 types of handlers:
1) Exec : Executes a sepcific commmand and is success if the status code is 0
2) TCP Socket Action : checks if a TCP port is open.
3) HTTP Get Action : Get requets on a predefined path and the container ip address. If status 200 then success

Liveness probe : done to check if a container is up and running. If there is a failure status restart policy for the container kicks in.
Readiness probe : done to check if the pod is ready to service requests. If this is present then request will flow only after readiness probe returns success after initialDelay. The default status is FAILED
StartUp Probe : Startup probe to check if the container has started. All other probes wait to execute till the StartUp Pod returns success.

Restart policy can be of 3 states , i,e , Always (Default) , OnFailure, Never. Restart policy is applicable to the container

Pod prioirty can be set to evict or pre-empt lower prioirty pods to make space for higher prioirty pods. Pre-empted pod gets graceful termination
A prioirty class object needs to be cretaed and then the prioirty class need to be specified on the pod yaml.

Replica Sets and Replica Controllers

REPLICA SET

Replica set scale and heal.
Replica sets wraps pod template and number of replicas, while deployment wraps repica ser within it, versioning and roll back
Controller for each type of objecy. They run reconciltioan loops to check expected and actual state
Same concept is used for replica set controllers.

ReplicaSet contains a template for a Pod. There is a selector on the pod which helps replica set in managing the pods.
Alternatives to replicaSet
- Deployment (versioning and rollback)
- Run to completeion Jobs: Batch processing (for pods which need to get terminated once work is done)
- DeamonSet: Pod lifetime tied to node

```
kind: ReplicaSet
spec:
    selector:
        matchLabels:
            <key> : <value>
            ---
            ---
            ---
    template: ------ pod specification begins
        metadata:
        spec:
            conatiner:
            - name:
              image:
              ---



Sample replica set yaml
              ---
```

<u>Working with replica set</u> :

1. Delete

kubectl delete <replicaSet> ---- scale rs to 0 and delete the pod
kubectl delete --cascade=false ---- deleet just the rs and not the pods. The pods are orphanded and are vulnerable

2. Isolation

change labels on the pod, resulting in the selector lavbels in RS will not match. But the RS will replace with new ones as RS notices that the number of Pods have decreased

3. Scale

change the replica set value in yaml and apply the RS yaml

4. Auto-scaling

An object HorizontalPodAutoScaler is required, which will be having the policy for sclaing. FOr eg based on CPU usage
-- a control loop to track actial and desired CPU utilization
Wont work with objects that cannot be scaled (DeamonSet)

<u>Loose Coupling in RS</u>

A coupling is loose because :

- RS can be deleted without afecting the pod
- Can isolate pods from RS by renaming the label on pod

<u>REPLICATION CONTROLLER</u>

Deprecated and outdated. similar to RS and deployments.
RC supports only equality based selector support while RS uses set-based

DEPLOYMENTS

Deployment encapsulates RS, provides versioning and rollback

RS associated with deployment :

Container in pod template is created
New RS and new Pods are created
Old RS continues to exist but gradually reduced to zero.

Rollback :

Every change to depliyment is tracked by kubernetes
creates a new revision of deployment
easy to roll back
new revision of deployment is created with every change to pod template
if changes are made, which do not correspond to pod template then no new revision is created
Only the pod deployment part is rolled back. Everything else remains same

During rolback, 2 versions of RS exist. the new one and old one

kubectl rollout undo deployment/deplyment-name
kubectl rollout undo deployment/deplyment-name --to-version=rollback

Usecases of Deployments :

- to rollout new RS
- updare existing deployment by updating the pod template. once apply is executed, new RS are created and pods move to new RS in a controlled manner
- rollback to previous version
- scale up
- Pause/resume deployment midway
- check status of deployment
- cleanup old RS

Fields in Deployment YAML

- selector
- Strategy (how old pods are replaced)
  * Recreate
  * RollingUpdate
- rooBackTo
- minReadySec (to keep the container ready for minimum time before declaring the pod ready)

```
kubectl create -f <deployment> --record  >>>>>>>>>>> record records eevrything in kubernetes


apiVersion: v1
kind: Deployment
metadata:
    name: <name>
    label:
        key: value
spec:
    selector:
        matchLabels:
            key:value
        template:                      >>>>>>>>>>>>>>>>>> the template for the pod
            metadata:
            spec:
                template:
                    conatiner:
                     - name:
                       image:


sample deployment template
```

Resume/Pause Deployment

kubectl rollout pause depoyment/deployment-name
kubectl rollout resume depoyment/deployment-name

if deployment pauses, changes in pod template will not create a new revision
Cannot be rolled back till deployment is complete

Clean Up Policy

revisionHistoryLimit -- controls how many revisions are kept.

Other Controllers

Stateful Sets

Help in managing Pod
Maintains a sticky identity for each pod meaning pod are not interchangeable and maintain a unique id

UseCases:
- Ordered graceful deployment, scaling, deletion and termination
- Ordered automated rolling update
- Stable unique network identifier and persistent storage

Guarantees:
- Sequence of pod creation and deltion is mantained
- All predecessors must be running before scaling up
- All successors must be deleted before deleting a pod

```
kind: StateFulSte
metadate:
    name:web
spec:
    serviceName: nginx
    replicas: 2
    selector:
        matchLabesl:
            app: nginx
     template:
        metadata:
            labels:
                app: nginx


On applying
pod will be created with names web-01, web-02
```

<u>DaemonSet</u>

Ensure all or some subset of nodes run a copy of a pod
As nodes are added, pods are created, and nodes are deleted, pods are garbage deleted

UseCases:

Cluster Storage Deamons
log collections deamons
node monitoring deamon

Static pod: pods not created by kubectl but by writing to directory watched by kubelet

<u>Run To Completion:</u>

create pods, track their completion
Ensure specifred number terminate succesfuly and then delete the pod marking the job complete

Once completed, no pods are created.
Pods are not deleted but state is terminated

Types:

Non-Parallel Job: force 1 pod ti run succesfully
Parallel jobs: job completed when number of completion reaches target
parallel jobs with work queues : requires coordination between pods

<u>Cron Jobs</u>

Similar to crontab
Schedule a job to be repeatedly executed at a point in time

Services

Containers need to expose ports in Pod spec, but Pod ip addresses keep changing.
Service: Logical set of pods + stable front end

front end: static ip address + port + dns

Service can be associated with any number of Pods based on label selector, forming a loose coupling between services and pods

Dynamic list of pods that are selected by a Service
Each service object has an associated endpoint object
Kubernetes evaluates service label selector vs all pods in the cluster. This list is dynamic in nature

Each service object must be associated with the endpoint object which does the dynamic updation of the list of pods associated with the service.

<u>VIrtual IP Access to Service</u>

can be used by clients outside cluster to access service object
Implemented by kube proxy which runson each node of cluster
Each kube-proxy will relay external traffic to correct virtual IP

<u>Types of Services</u>

* *ClusterIP :*

    Static lifetime IP of service
    Service accessible within cluster
    ClusterIP is independent of backend pods
    Default type of service

* *Nodeport :*

    Service exposed on each node on static port
    Extenral clients can hit node ip +node port
    Request will be relayed tp cluster IP + nodeport (the mapping from node ip to cluster ip is done by kube-proxy)

* *LoadBalancer :*

    External LB provided by cloud
    Will create node port and cluster ip
    External LB -> Node IP -> Cluster IP

* *ExternalName :*

    Map service to extenal service residing out of cluster
    configured and managed by kube-dns.
    No selector or port.

How of kubernetes networking?

Pod always communicate with each other
Inter-pod communication independednt of nodes
Pods have private IP within cluster and are reachabel to each other without DNS
Containers withing a pod can be accesed via localhost.

When service object is created a cluster IP is assigned, which is independent of pods and lives as long as the service object
ervice object has a static port assigned, which is reserved for the service throughout the cluster

Virtual IPs Service

Virtual IP address can be relayed and translated clusterIP by any node

Kube-proxy does this trnaslation

Service Discovery:

- DNS loopup
  - ◇ Required DNS add on. DNS server listensts creation of new service object.
  - ◇ once created dns records are created
  - ◇ All nodes can resolve using the dns lookup and name

- Environment Variable
  - ◇ Each service has environment variable for host and port
  - ◇ Static, not updated after pod creation. So the Service should be the first object created

Headless Service

No clusterIP, no load balancing.
Kube-proxy will not work for headless service

NodePort Service

service exposed on each node on static port
External clients can hit node ip + node port which is re-routed to cluster ip + node port by the kube-proxy

LoadBalancer Service

- Service.Type = LoadBalancer
  - ◇ Use LBs provided by a cloud provider. Tightly coupled to cloud provider.
  - ◇ This is managed by the cloud controller manager

- Ingress object

DNS for Services

When service created , DNS server creates name
if pods are in different namesapce then the fully qualified can be used