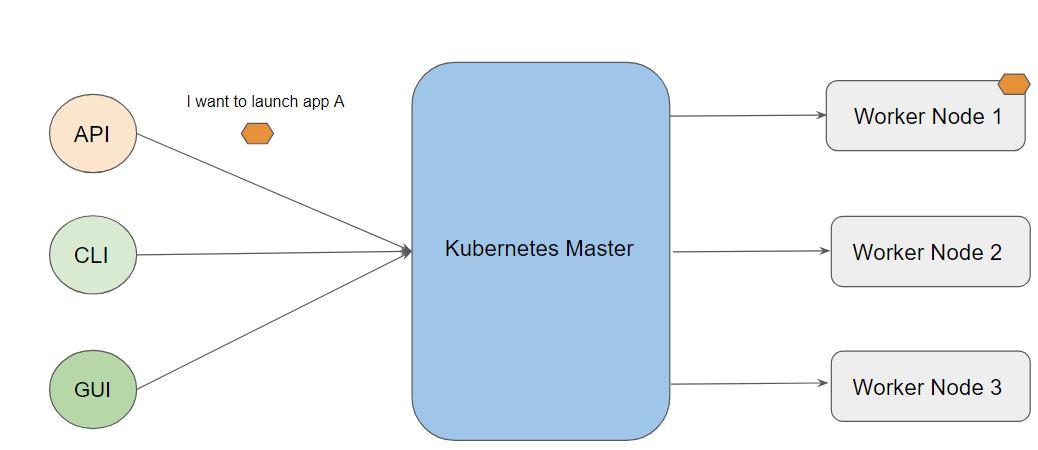
**Introduction to Kubernetes**

Kubernetes (K8s) is an open-source container orchestration engine developed by Google.

It was originally designed by Google and is now maintained by the Cloud Native Computing Foundation.



**Installation Options for Kubernetes**

There are multiple ways to get started with a fully functional Kubernetes environment

1. Use the Managed Kubernetes Service (GKE, EKS, AKE, OKE, Tanzu, Openshift)
2. Use Minikube
3. Install & Configure Kubernetes Manually (Hard Way)
4. Kubeconfg and kubeadm
5. Kops
6. rke

Managed Kubernetes Service

Various providers like AWS, IBM, GCP, and others provide managed Kubernetes clusters.

Most organizations prefer to make use of this approach.



Minikube

Minikube is a tool that makes it easy to run Kubernetes locally.

Minikube runs a single-node Kubernetes cluster inside a Virtual Machine (VM) on your laptop for users looking to try out Kubernetes or develop with it day-to-day.



**Official Documentation:**

<https://kubernetes.io/docs/tasks/tools/install-minikube/>

Play Ground:

<https://labs.play-with-k8s.com/>

Steps to install K8s using Minikube:

1.Take Ubuntu AWS machine with t2.medium or t2.large

Note: please run all commands using ubuntu suer only

2. Install docker sudo apt update;sudo apt install docker.io

3. enable and start docker (sudo systemctl enable docker;sudo systemctl start docker)

3. Add ubuntu user in docker group sudo usermod -aG docker $USER && newgrp docker

4. After that refere below document and follow with ubuntu user only

<https://kubernetes.io/docs/tasks/tools/install-minikube/>

5. Configure kubectl

$curl -LO "https://dl.k8s.io/release/$(curl -L -s <https://dl.k8s.io/release/stable.txt>)/bin/linux/amd64/kubectl"

$sudo cp kubectl /usr/local/bin

$sudo chmod +x /usr/local/bin/kubectl

$kubectl get nodes

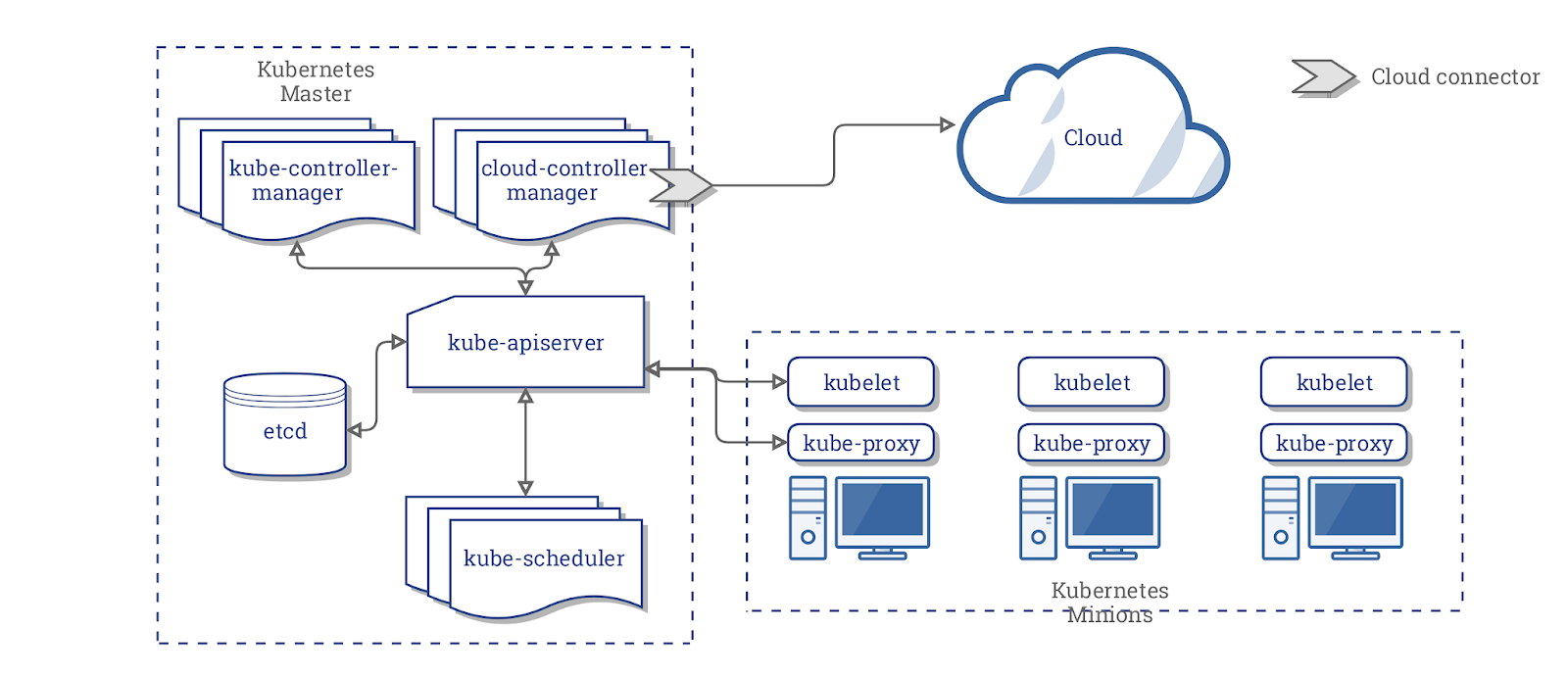
ubuntu@ip-172-31-47-15:~$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

minikube Ready control-plane 8m38s v1.26.1

Kubernetes the Hard Way

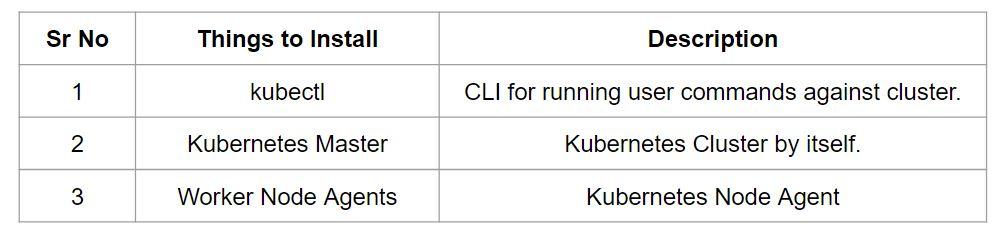
In this approach, you install and configure components of Kubernetes individually.



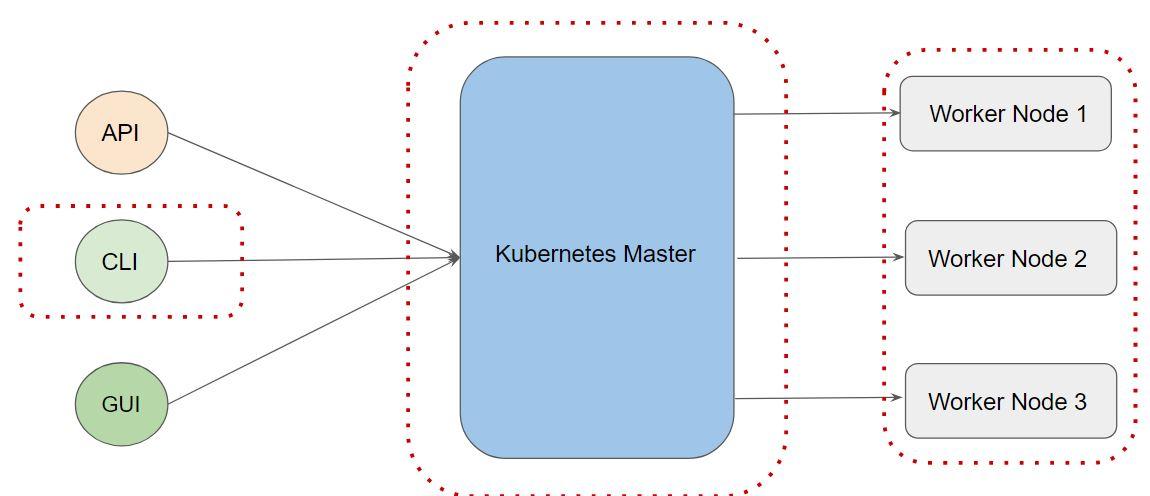
[ddometita/mmumshad-kubernetes-the-hard-way (github.com)](https://github.com/ddometita/mmumshad-kubernetes-the-hard-way)

Installation Configuration:

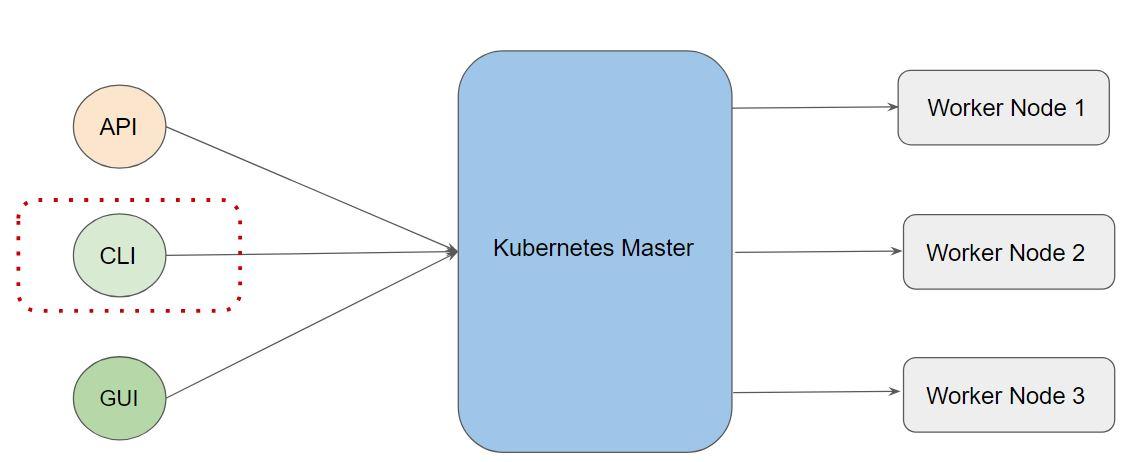
Things to configure while working with Kubernetes.



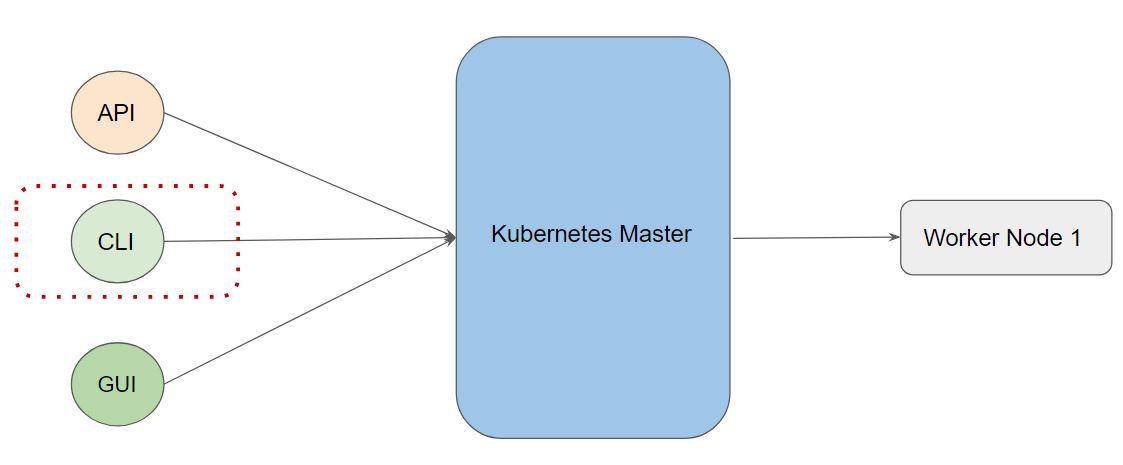
The following components highlighted in red are the ones that need to be configured while designing Kubernetes cluster in a hard way.



The following components highlighted in red are the ones that need to be configured while using managed Kubernetes cluster.



The following components highlighted in red are the ones that need to be configured while using minkube based installation method.

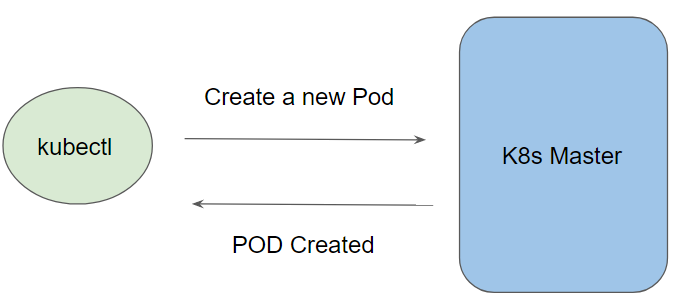


Note: installation can be followed by the which I shared in other notepad.

**Overview of kubectl**

The Kubernetes command-line tool, kubectl, allows you to run commands against Kubernetes clusters.

You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs.

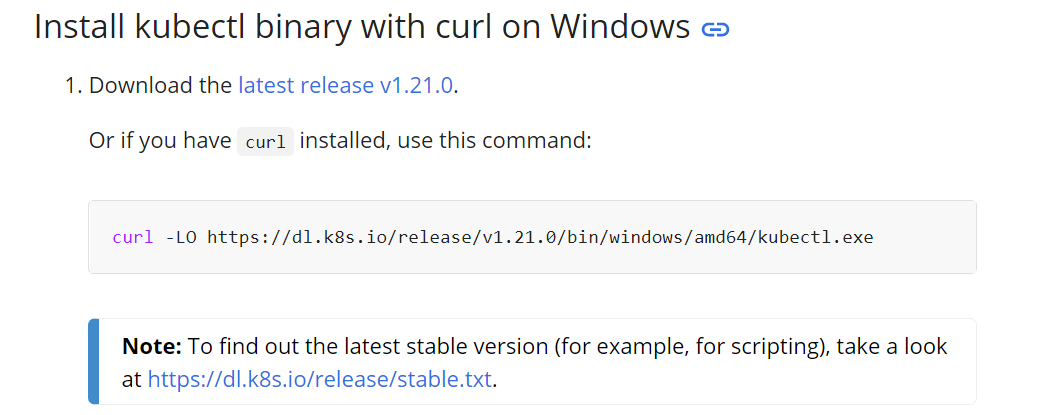


Let us understand the high-level workflow:

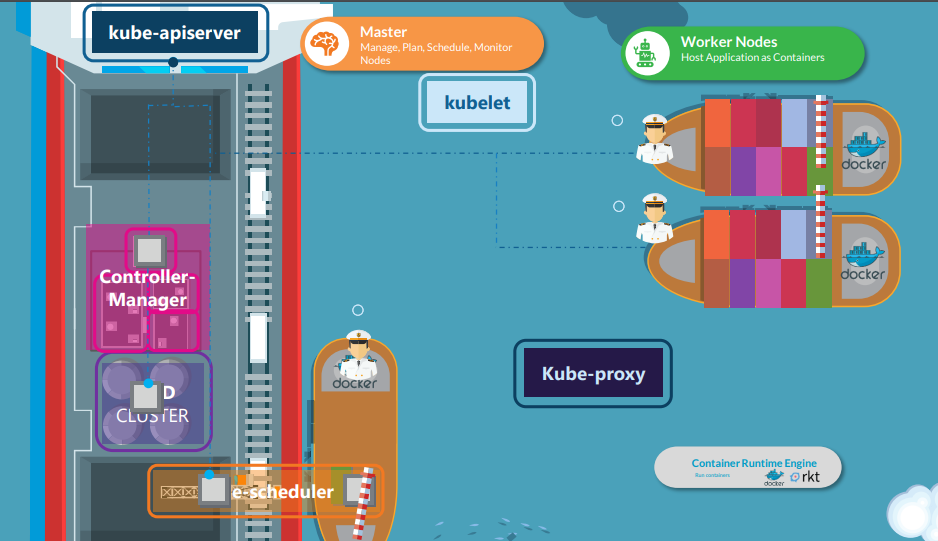
To connect to the Kubernetes Master, there are two important data which kubectl needs:

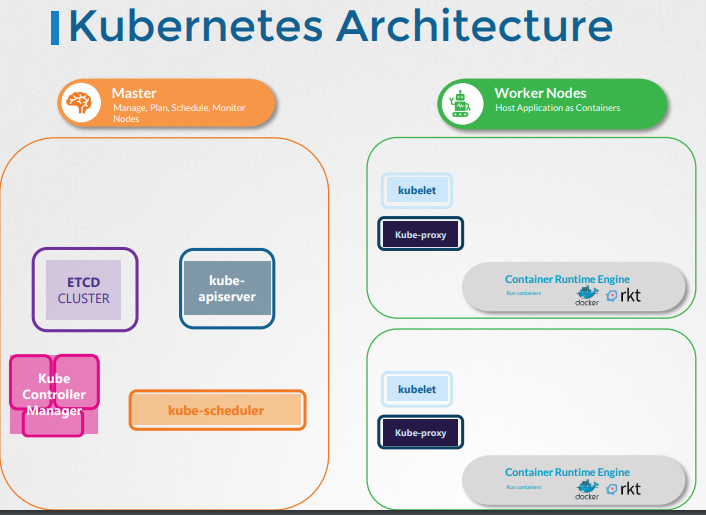
* DNS / IP of the Cluster
* Authentication Credentials

The overall installation of kubectl is straightforward and can be installed on a variety of Linux platforms, macOS and Windows



**Kubernetes Architecture:**





Flow Diagram:

1. Kubectl writes to the API server (kubectl run mywebserver --image=nginx)

2. API server will authenticate and authorize. Upon validation, it will write it to etcd.

3. Upon write to etcd, API Server will invoke the scheduler.

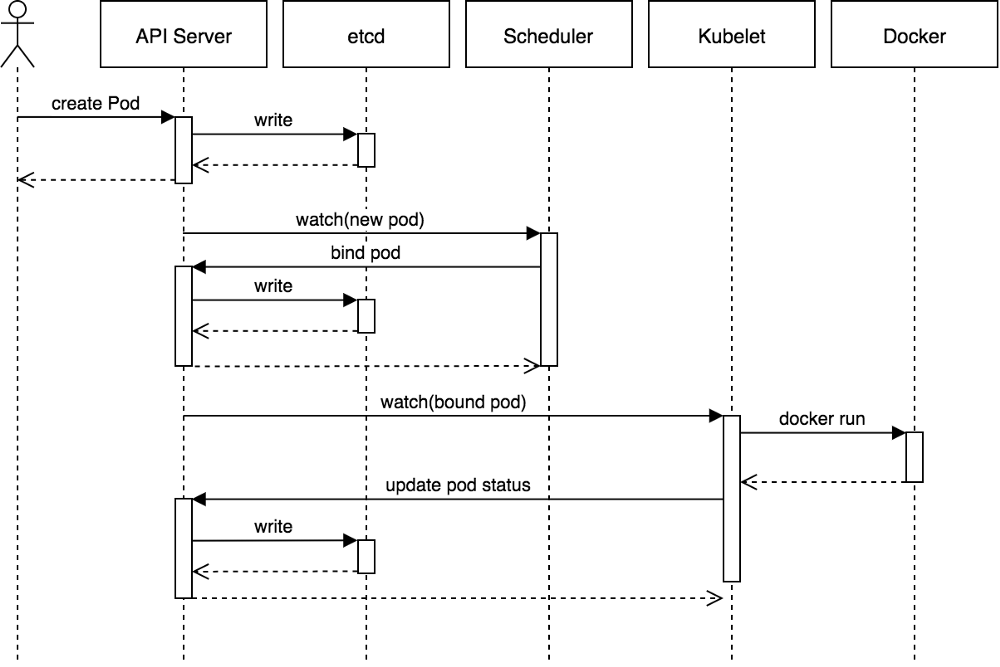
4. Scheduler decides which node the pod should run and return data to API server. API will in-turn write it back to etcd.

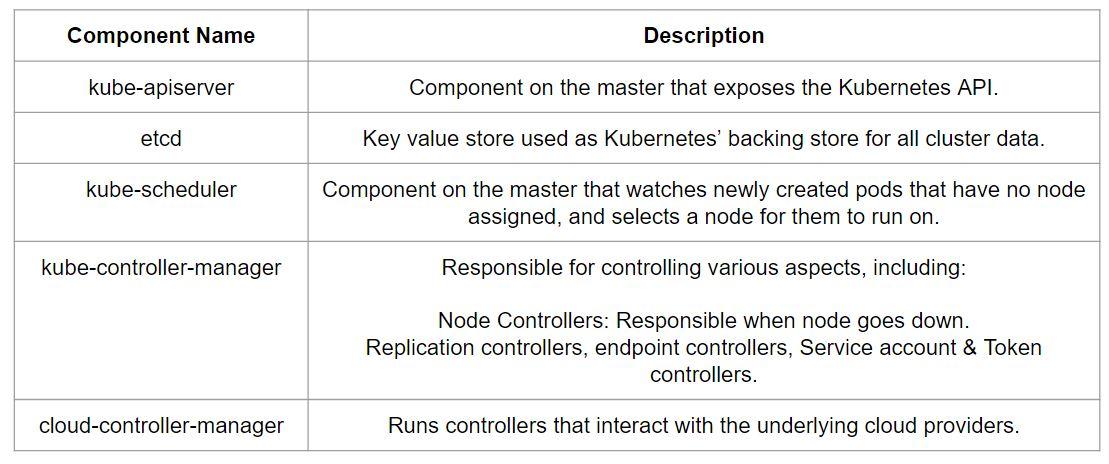
5. API Server will invoke the kubelet in the node decided by the scheduler.

6. Kubelet communicates to the docker daemon via Docker socket to create the container.

7. Kubelet will update the status of the POD back to the API server.

8. API Server will write the status details back to etcd.



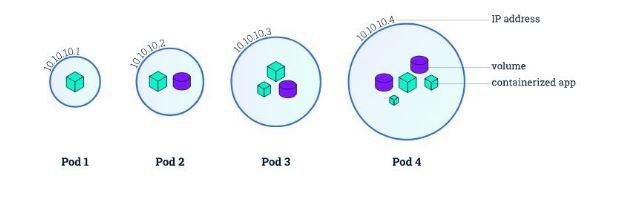


Node components run on every worker node of the Kubernetes Cluster.



**PODS**

A Pod in Kubernetes represents a group of one or more application containers and some shared resources for those containers.



Containers within a Pod share an IP address and port space and can find each other via the localhost.

A Pod always runs on a Node.

A Node is a worker machine in Kubernetes.

Each Node is managed by the Master.

A Node can have multiple pods.

**Kubernetes Object**

Kubernetes Objects is basically a record of intent that you pass on to the Kubernetes cluster.

Once you create the object, the Kubernetes system will constantly work to ensure that object exists.

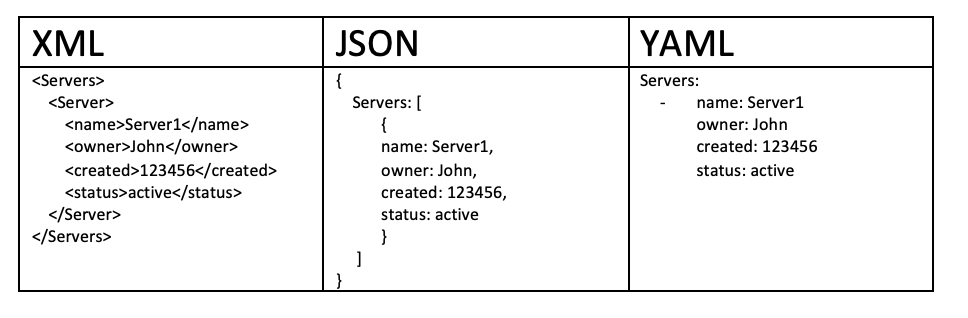
There are various ways in which we can configure a Kubernetes Object.

* The first approach is through the kubectl commands.
* The second approach is through a configuration file written in YAML.



YAML is a human-readable data-serialization language.

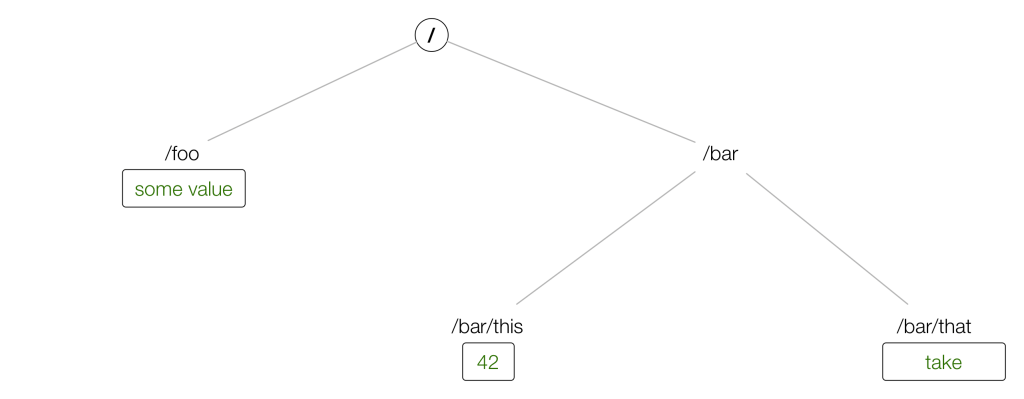
It designed to be human friendly and works perfectly with other programming languages.



**K8s Component - ETCD**

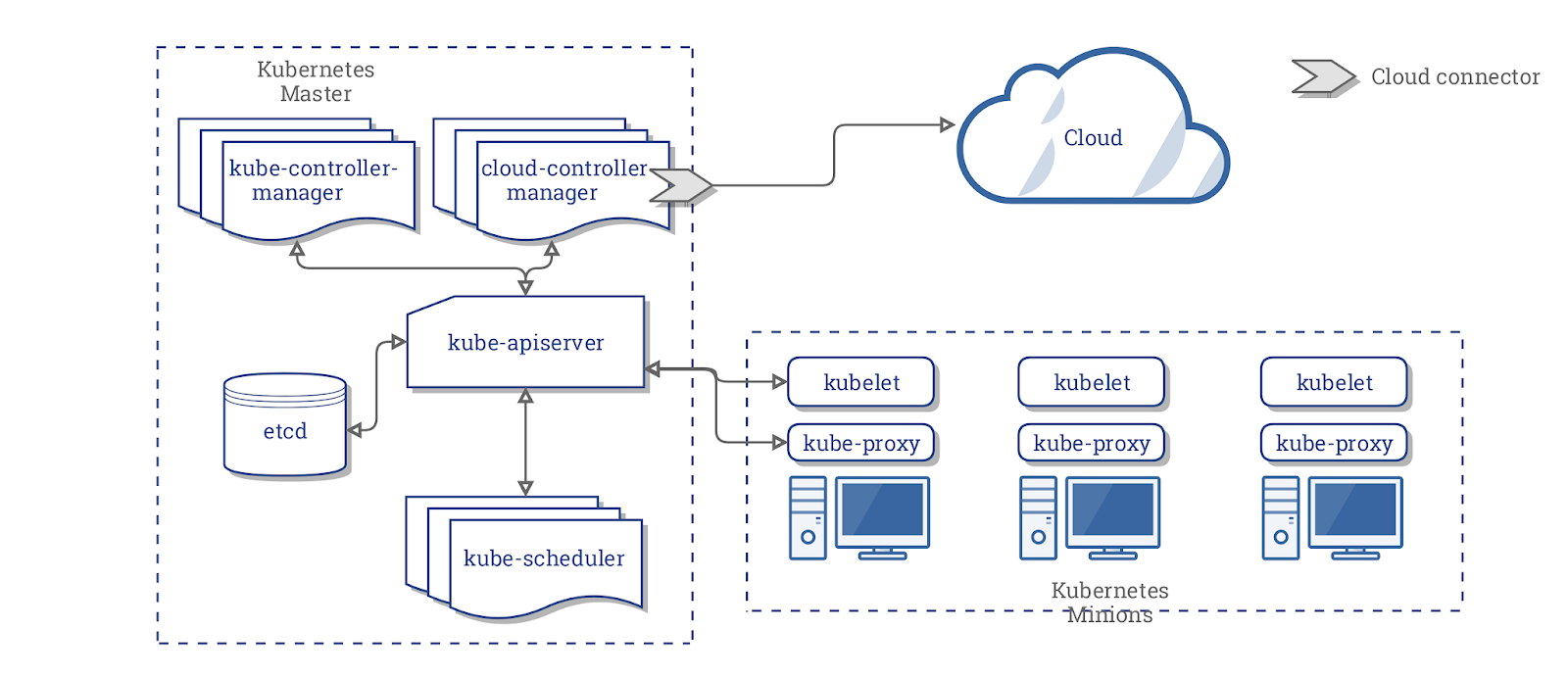
In a Linux environment, all the configurations are stored in the /etc directory.

etcd is inspired from that and there is an addition of d which is for distributed.



etcd is a distributed reliable key-value store.

etcd reliably stores the configuration data of the Kubernetes cluster, representing the state of the cluster (what nodes exist in the cluster, what pods should be running, which nodes they are running on, and a whole lot more) at any given point of time.



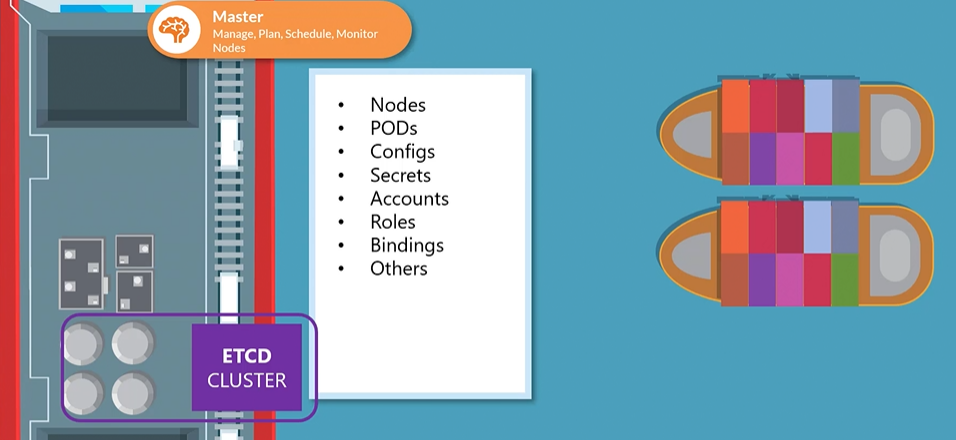
Important Pointers for ETCD

A Kubernetes cluster stores all its data in etcd.

Anything that you read while running kubectl get pods is stored in etcd

Any node crashing or process dying causes values in etcd to be changed.

Whenever you create something with kubectl create / kubectl run will create an entry in the etcd.



**K8s Component - kube-api server**

API server acts as a gateway to the Kubernetes Cluster.

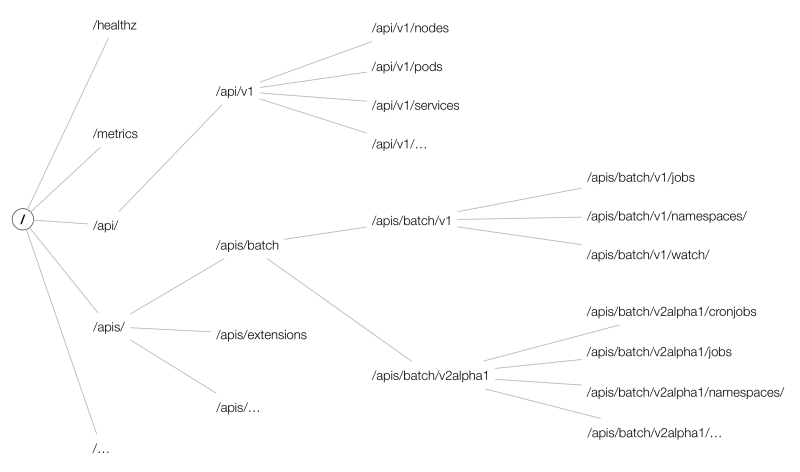
When you interact with your Kubernetes cluster using the kubectl command-line interface, you are actually communicating with the master API Server component.

The API Server is the only Kubernetes component that connects to etcd; all the other components must go through the API Server to work with the cluster state.

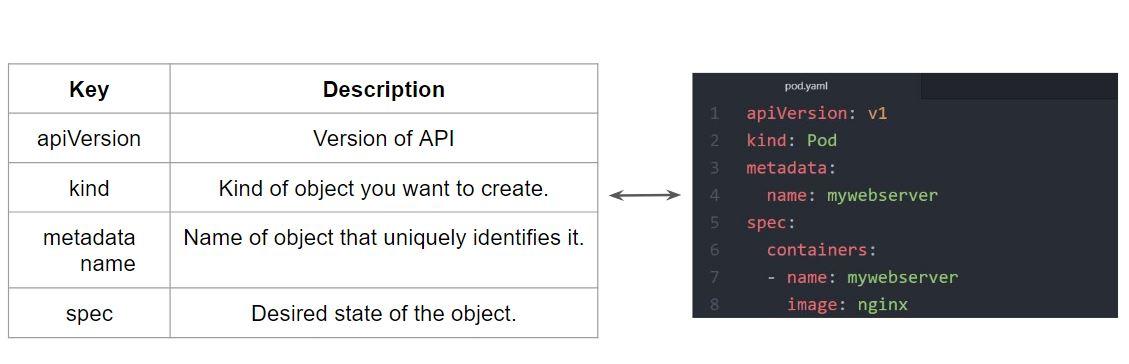
The API Server is also responsible for the authentication and authorization mechanism. All API clients should be authenticated in order to interact with the API Server.

**Kubernetes API Primitives**

Depending on the operation you intend to do, there are various APIs that are available.

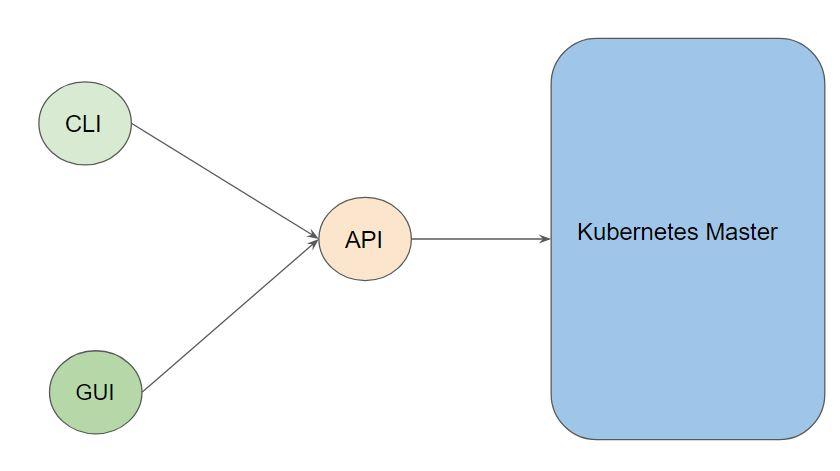


**Creating First POD Configuration in YAML**

****

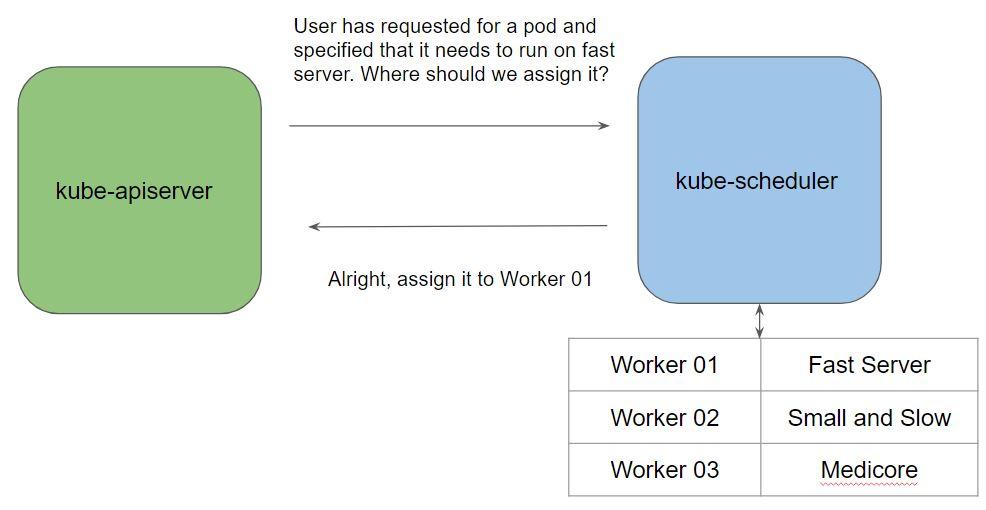


Following is a high-level architecture of a data flow in Kubernetes



**K8s Component - kube-scheduler**

Kube-scheduler watches for newly created pods that have no node assigned, and selects a node for them to run on.



There are several factors which are taken into consideration before a pod is scheduled to a node.

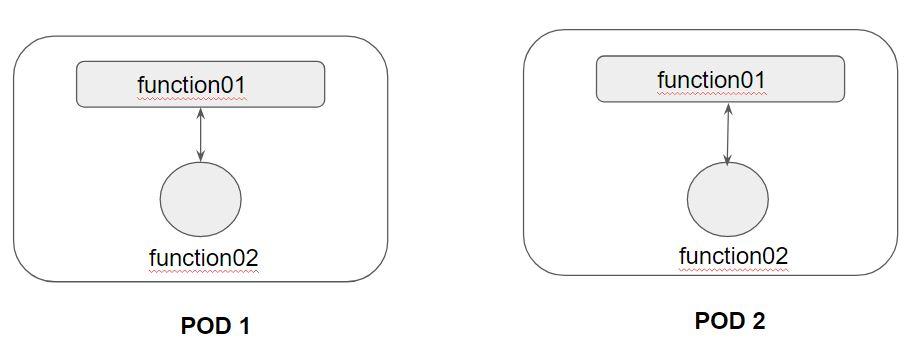
Some of these includes:

* Resource Requirements
* Hardware/Software policy constraints
* Affinity & Anti-Affinity
* Data Locality

**Multi-Container Pods**

A single POD can have multiple containers as part of it. Till now we have been working based on a single container per pod.

Containers within a Pod share an IP address and port space and can find each other via localhost.



apiVersion: v1

kind: Pod

metadata:

name: multicontpod

spec:

containers:

- image : nginx

name: container1

- image: busybox

name: container2

command:

- sleep

- "3600"

**Revising Dockerfile - CMD vs ENTRYPOINT**

The best use for ENTRYPOINT is to set the image’s main command

ENTRYPOINT doesn’t allow you to override the command.

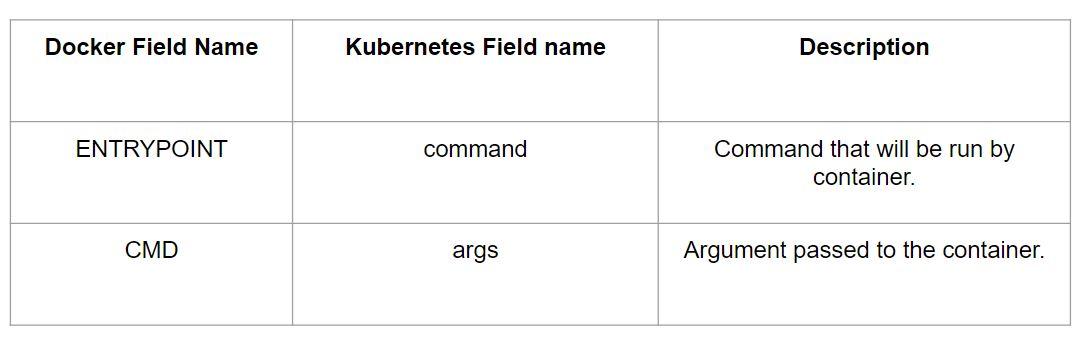
It is important to understand distinction between CMD and ENTRYPOINT.

**Command and Arguments**

During the video of ENTRYPOINT, we discussed the difference between CMD and ENTRYPOINT instruction in Dockerfile.

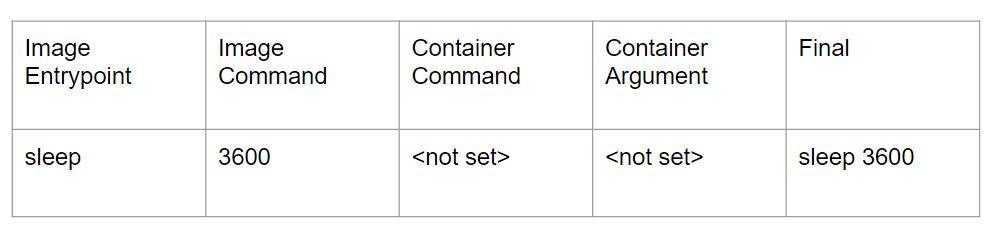
We can also refer them as Image Command and Image Entrypoint.

In Kubernetes, we can override the default entrypoint and CMD with command and args field.



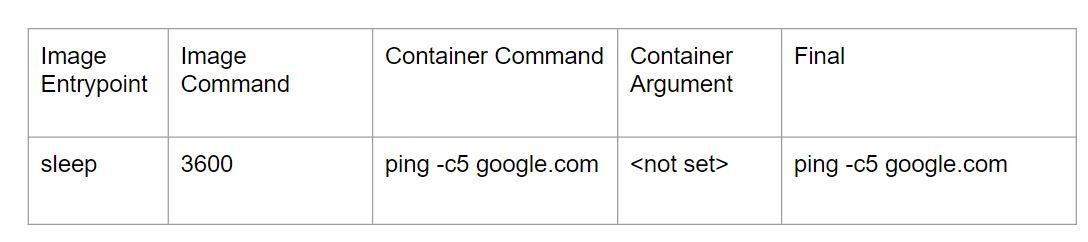
Use-Case 1: Image Entrypoint and CMD

When there is an entrypoint and CMD set for a Docker image and if we are not manually overriding it at k8s manifest level, then final decision is to use the default image specifications.



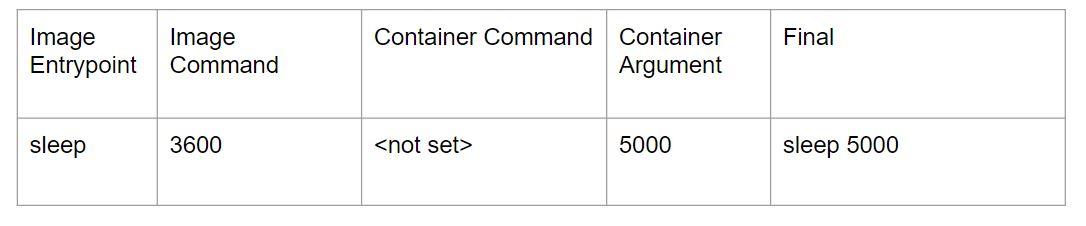
Use-Case 2: Setting Container Command

When there is an entrypoint and CMD set for a Docker image and if we are not manually overriding it at k8s manifest level, then final decision is to use the default image specifications.



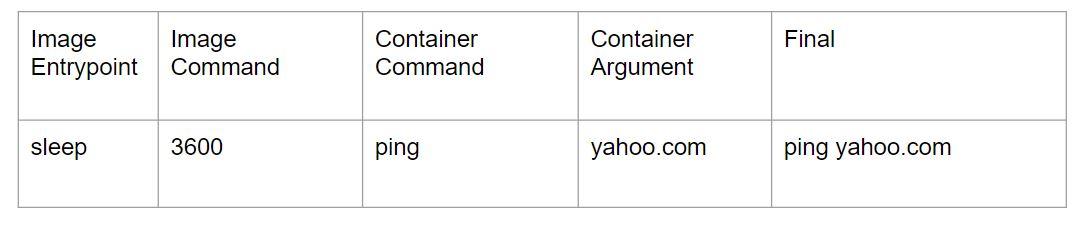
Use-Case 3: Setting Container Arguments

When container argument is set in k8s manifest, the image command gets overridden.



Use-Case 4: Setting Container Command & Arguments

When container command and arguments are specified in k8s manifest, they will override the image command and entrypoint.



**CLI Documentation of K8s Resources**

Till now we have been going through documentation from browser to understand about fields.

There is better way through which we can achieve similar functionality via CLI

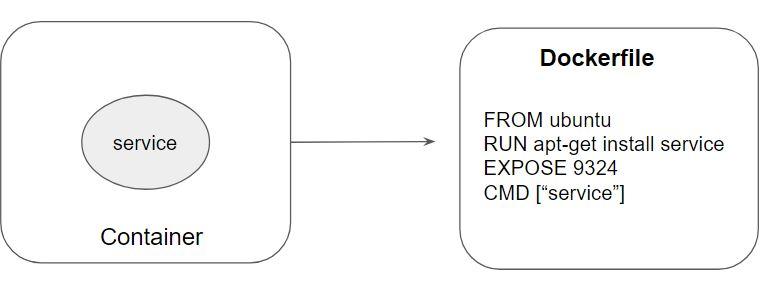
This is through the kubectl explain command.

**Revising DockerFile - EXPOSE Instruction**

The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime.

The EXPOSE instruction does not actually publish the port.

It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published.



**Labels & Selector**

Labels

Labels are key/value pairs that are attached to objects, such as pods

Selectors

Selectors allow us to filter objects based on labels.

Example:

Show me all the objects which have a label where env: prod



kubectl label pods <podname> <label>

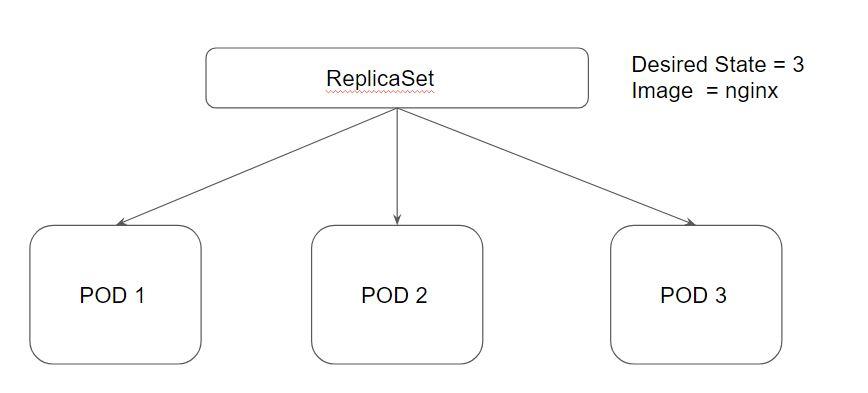
kubectl get pods --show-labels

kubectl get pods -l env=prod

kubectl get pods -l env!=prod

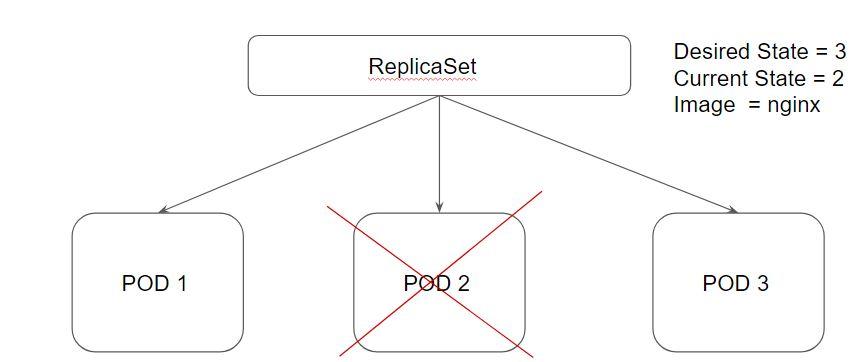
**ReplicaSets**

A ReplicaSet purpose is to maintain a stable set of replica Pods running at any given time.



Desired State - The state of pods which is desired.

Current State - The actual state of pods that are running.

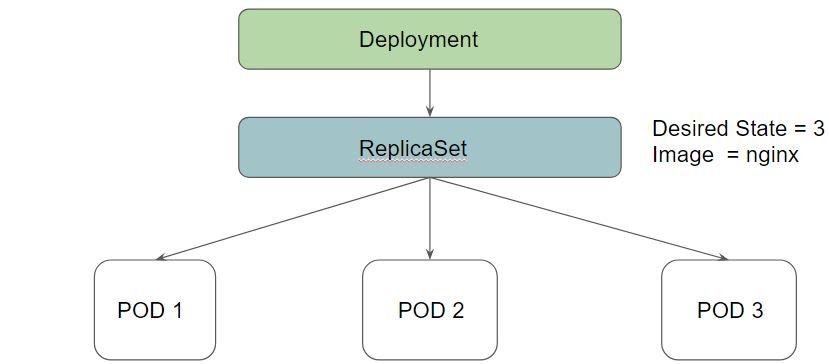


kubectl scale --replicas=2 rs/frontend

kubectl autoscale rs frontend --max=10 --min=3 --cpu-percent=50

**Deployments**

Deployments provide replication functionality with the help of ReplicaSets, along with various additional capability like rolling out of changes, rollback changes if required.



Benefits of Deployment - Rollout Changes

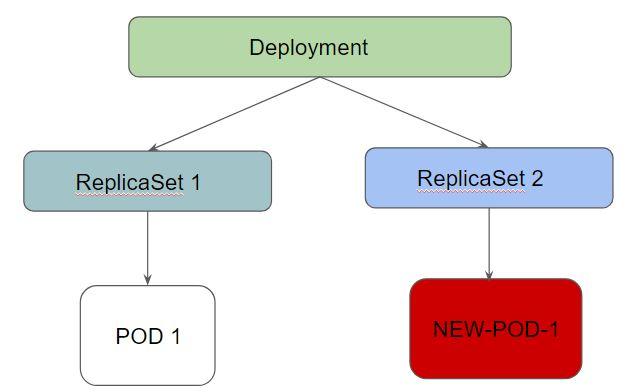
We can easily roll out new updates to our application using deployments.

Deployments will perform an update in a rollout manner to ensure that your app is not down.



Benefits of Deployment - Rollback Changes

Sometimes, you may want to rollback a Deployment; for example, when the Deployment is not stable, such as crash looping



Deployment ensures that only a certain number of Pods are down while they are being updated.

By default, it ensures that at least 25% of the desired number of Pods are up (25% max unavailable).

Deployments keep the history of revision which had been made.

958 kubectl set image deployment/frontend php-redis=gcr.io/google\_samples/gb-frontend:v3 --record

959 kubectl set image deployment/frontend php-redis=gcr.io/google\_samples/gb-frontend:v3 --record

960 kubectl rollout history deployment frontend

961 k get rs

962 clear

963 k delete deployment frontend

964 cat deployment.yaml

965 k get deployment

966 k get pods

967 k get rs

968 k apply -f deployment.yaml

969 clear

970 k get deployment

971 k get rs

972 k get pods

973 vi deployment.yaml

974 kubectl set image deployment/frontend php-redis=gcr.io/google\_samples/gb-frontend:v3 --record

975 k get rs

976 kubectl rollout history deployment frontend

977 k get pods

978 vi deployment.yaml

979 k apply -f deployment.yaml

980 k get pods

981 k get deployment.yaml

982 k get rs

983 k get pods

984 kubectl rollout history deployment frontend

985 cat deployment.yaml

986 vi deployment.yaml

987 k get pods

988 k describe pod frontend-5f9677ff88-8vkzw

989 k apply -f deployment.yaml

990 k get pods

991 k get rs

992 k get pods

993 k describe pod frontend-c646cdf7f-zw4dc

994 kubectl rollout history deployment frontend

995 k describe pod frontend-c646cdf7f-zw4dc

996 k get pods

997 k get rs

998 k get pods

999 k describe pod frontend-c646cdf7f-6cnl7

1000 kubectl rollout history deployment frontend

1001 kubectl rollout undo deployment frontend

1002 kubectl rollout history deployment frontend

1003 k get pods

1004 k describe pod frontend-5f9677ff88-78jd6

1005 kubectl rollout history deployment frontend

1006 kubectl rollout undo deployment frontend

1007 k get pods

1008 k describe pod frontend-c646cdf7f-29lzq

1009 kubectl rollout undo deployment frontend --to-revision=2

1010 k get pods

1011 k describe pod frontend-84855979d9-pwd9l

1012 kubectl rollout history deployment frontend

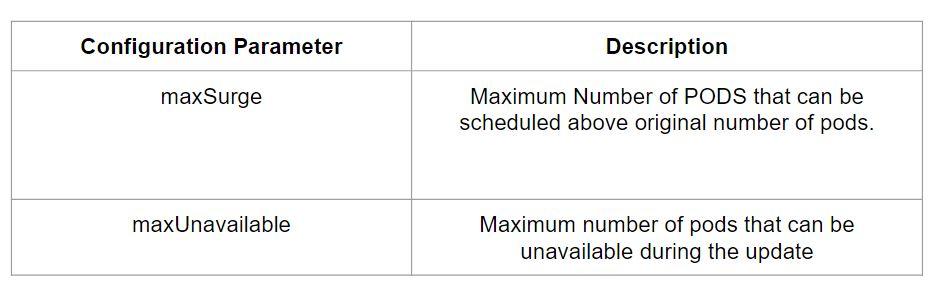
kubectl rollout undo deployment nginx-deployment --to-revision=2

Note: you can scale deployment using below command

kubectl scale deployment/nginx-deployment --replicas=10

**Deployment Configuration**

While performing a rolling update, there are two important configurations to know.



maxUnavailable=0 and maxSurge=20% << Full Capacity is maintained.

maxUnavailable=10% and maxSurge=0 << Update with no extra capacity. In-place updates.

If you want fast rollout, make use of maxSurge.

If there might be a resource quota in place and partial unavailability is acceptable, maxUnavailable can be used.

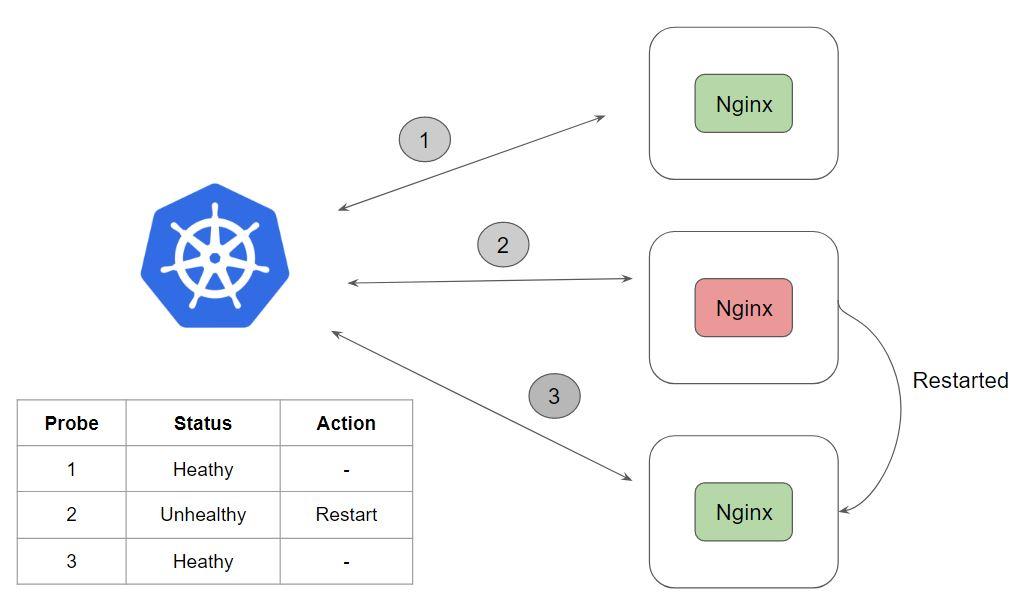
Deployment-strategies-in-kubernetes:

[Deployment Strategies In Kubernetes (auth0.com)](https://auth0.com/blog/deployment-strategies-in-kubernetes/)

**Liveness Probe**

Many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted.

Kubernetes provides liveness probes to detect and remedy such situations.



There are 3 types of probes which can be used with Liveness

* HTTP
* Command
* TCP

**apiVersion**: v1

**kind**: Pod

**metadata**:

**labels**:

**test**: liveness

**name**: liveness-exec

**spec**:

**containers**:

- **name**: liveness

**image**: k8s.gcr.io/busybox

**args**:

- /bin/sh

- -c

- touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600

**livenessProbe**:

**exec**:

**command**:

- cat

- /tmp/healthy

**initialDelaySeconds**: 5

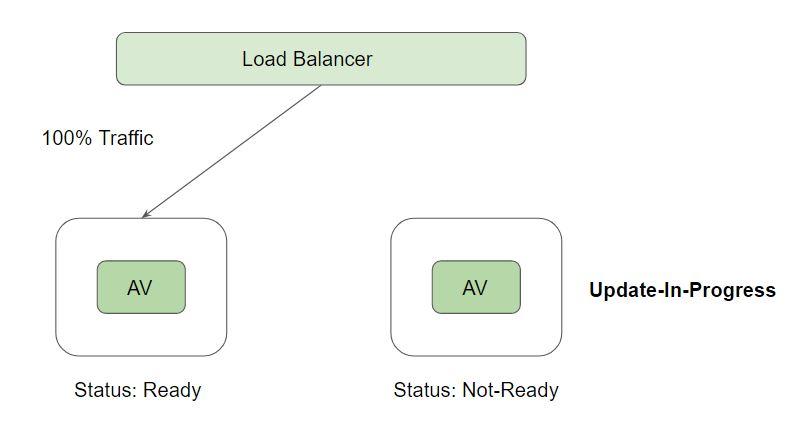
**periodSeconds**: 5

**Readiness Probe**

It can happen that an application is running but temporarily unavailable to serve traffic.

For example, the application is running but it is still loading it’s large configuration files from external vendors.

In such a case, we don’t want to kill the container however we also do not want it to serve the traffic.



root@kmaster:~# cat readyness.yaml

apiVersion: v1

kind: Pod

metadata:

labels:

test: liveness

name: liveness-exec

spec:

containers:

- name: liveness

image: k8s.gcr.io/busybox

args:

- /bin/sh

- -c

- sleep 20; touch /tmp/healthy; sleep 600

readynessProbe:

exec:

command:

- cat

- /tmp/healthy

initialDelaySeconds: 5

periodSeconds: 5

==========

root@kmaster:~# cat readyness.yaml

apiVersion: v1

kind: Pod

metadata:

labels:

test: liveness

name: liveness-exec

spec:

containers:

- name: liveness

image: k8s.gcr.io/busybox

args:

- /bin/sh

- -c

- sleep 30; touch /tmp/healthy; sleep 20; rm -f /tmp/healthy; sleep 600

readinessProbe:

exec:

command:

- cat

- /tmp/healthy

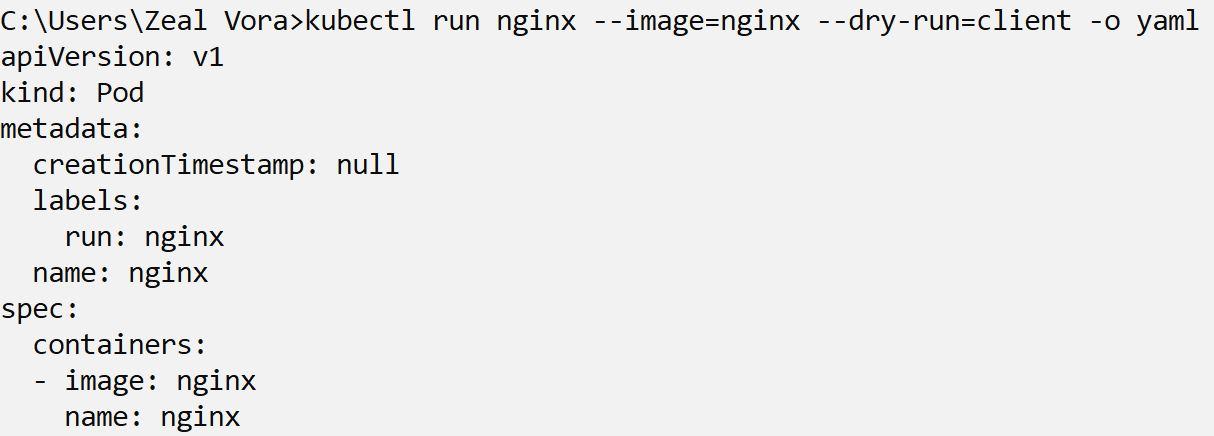
initialDelaySeconds: 3

periodSeconds: 3

**Generating Deployment Manifests via CLI**

Till now, we have been referencing the documentation to get the manifests for objects like Pods.

This can be a tedious process and can consume time.



#### **step 1. Creating Deployment via CLI**

kubectl create deployment DEPLOYMENT-NAME --image=[IMAGE-NAME]

#### **Step 2. Generating Deployment Manifest via CLI**

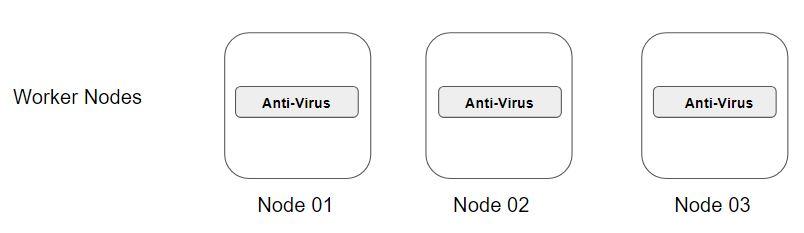
kubectl create deployment DEPLOYMENT-NAME --image=[IMAGE-NAME] --dry-run=client -o yaml

kubectl create deployment --help

**Daemonsets**

A DaemonSet can ensure that all Nodes run a copy of a Pod.

As nodes are added to the cluster, Pods are added to them.



apiVersion: apps/v1

kind: DaemonSet

metadata:

name: repotech-daemonset

spec:

selector:

matchLabels:

name: repotech-all-pods

template:

metadata:

labels:

name: repotech-all-pods

spec:

containers:

- name: repotech-pods

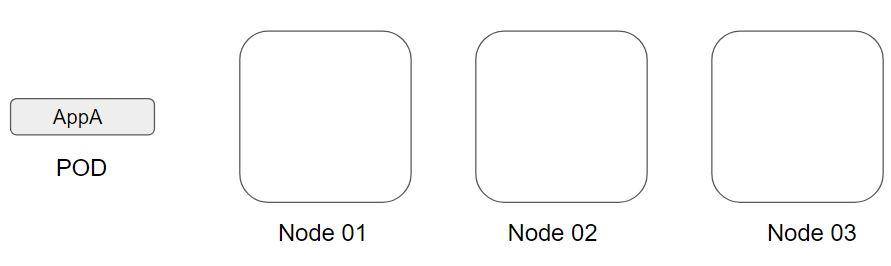
image: nginx

**NodeSelector**

nodeSelector allows us to add a constraint about running a pod in a specific worker node.

Use-Case:

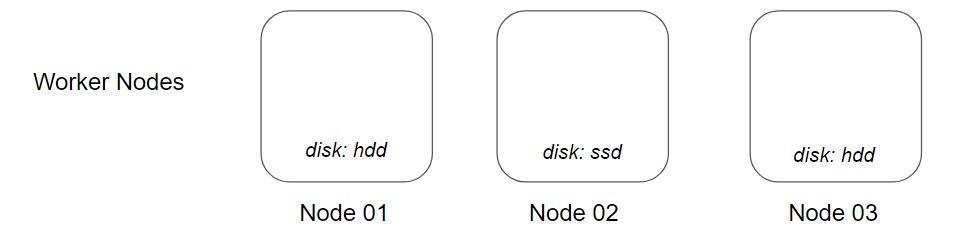
* AppA requires faster disk in order to be able to run effectively.
* Run AppA in nodes which has SSD.



Step 1: Adding a Label to the Node

Step 1: Add a Label to your nodes depending on their disk types.

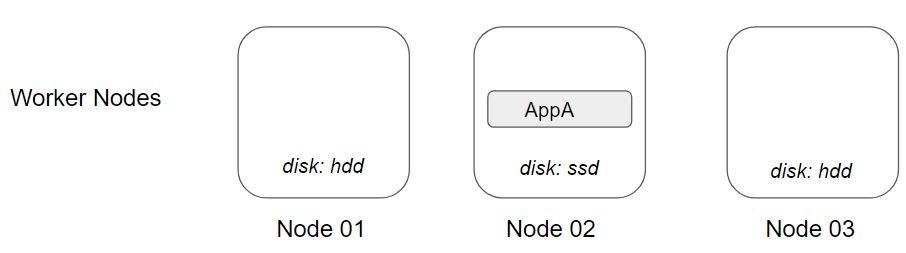
* disk: hdd
* disk:ssd



kubectl label node <name of node> key=value

Step 2: Using nodeSelector Configuration

Create a nodeSelector configuration to run pods only on nodes which has a label of disk=ssd



apiVersion: v1

kind: Pod

metadata:

name: service-pod

spec:

containers:

- name: service-pod

image: nginx

nodeSelector:

disk: ssd

**Node Affinity**

For multiple reasons, there can be a need to run a pod on a specific worker node.

There can be multiple reasons, node hardware being the common one.

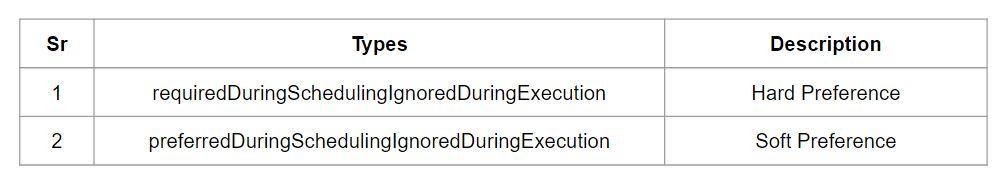
Node affinity is a set of rules used by the scheduler to determine where a pod can be placed

In Kubernetes terms, it is referred as nodeSelector, and nodeAffinity/podAffinity fields under PodSpec.

In Kubernetes, we can achieve nodeAffinity with the help of:

* nodeSelector
* nodeAffinity (more flexibility)

Node affinity is conceptually similar to nodeSelector – it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.



apiVersion: v1

kind: Pod

metadata:

name: node-affinity

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: disk

operator: In

values:

- ssd

containers:

- name: with-node-affinity

image: nginx

====

apiVersion: v1

kind: Pod

metadata:

name: node-affinity-preferred

spec:

affinity:

nodeAffinity:

preferredDuringSchedulingIgnoredDuringExecution:

- weight: 1

preference:

matchExpressions:

- key: memory

operator: In

values:

- high

- medium

containers:

- name: affinity-prefferd

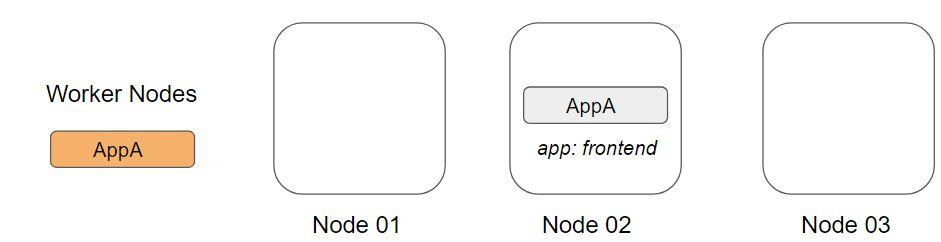
image: nginx

**Pod Affinity and Pod Anti-Affinity**

First Question: Where should I be running this pod?

With Node Affinity, the question became: Should I be running my pod in this node?T

The considerations are still about the node. No outside information is considered apart from node.

****

Step 1: Pod Selector

The first thing to figure out: “What other POD are we referring to”

In this case, we are referring to other POD as AppA which has a label of app:frontend

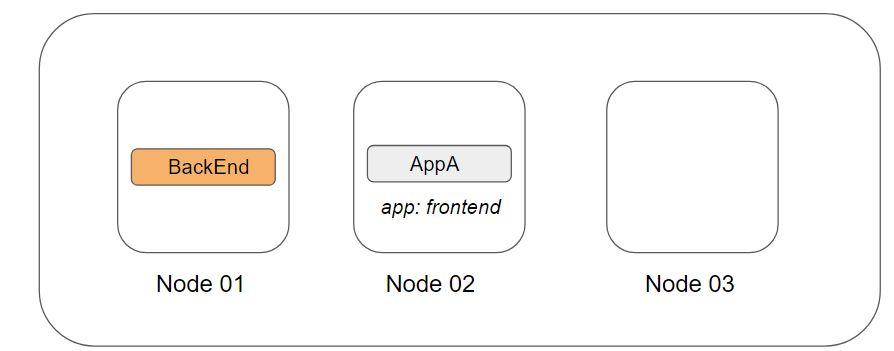
Answer: I want BackEnd Pod to be running in the same place as AppA Pod.

Step 2: Topology

Topology refers to “what does the same place mean”?

It can mean the same place if we look at the zone or region level (same AZ / same region)

It can mean a different place if we look at the host level.



Step 3: Yes/No

Should I run my pod in the same place as the other POD? (Yes or No)

Yes: Pod Affinity

No: Pod Anti-Affinity

===

kind: Pod

metadata:

name: pod-affinity

spec:

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: app

operator: NotIn

values:

- frontend

topologyKey: kubernetes.io/hostname

containers:

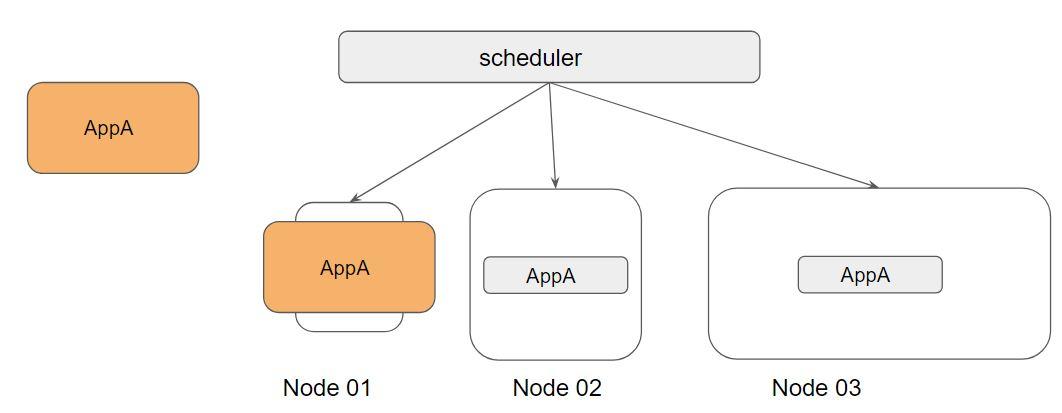
- name: pod-affinity

image: nginx

[Assigning Pods to Nodes | Kubernetes](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/)

**Resource Requests and Limits**

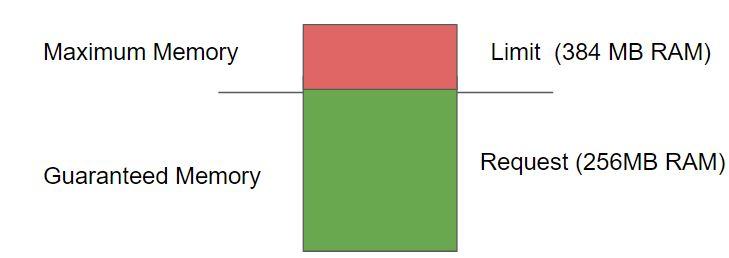
If you schedule a large application in a node which has limited resource, then it will soon lead to OOM or others and will lead to downtime.



Requests and Limits are two ways in which we can control the amount of resource that can be assigned to a pod (resource like CPU and Memory)

Requests: Guaranteed to get.

Limits: Makes sure that the container does not take node resources above a specific value.



Kubernetes Scheduler decides the ideal node to run the pod depending on the requests and limits.

If your POD requires 8GB of RAM, however, there are no nodes within your cluster which has 8GB RAM, then your pod will never get scheduled.



apiVersion: v1

kind: Pod

metadata:

name: kplabs-pod

spec:

containers:

- name: kplabs-container

image: nginx

resources:

requests:

memory: "64Mi"

cpu: "0.5"

limits:

memory: "128Mi"

cpu: "1"

- name: kplabs-container1

image: busybox

resources:

requests:

memory: "64Mi"

cpu: "0.5"

limits:

memory: "128Mi"

cpu: "1"

**Static Pods**

You can directly inform the kubelet that it needs to run a specific pod.

There are multiple ways in which you can tell kubelet to run a pod.

Pod created directly without schedulers are also referred as Static Pods.

systemctl status kubelet

cd /etc/systemd/system/kubelet.service.d

===

apiVersion: v1

kind: Pod

metadata:

name: repotech-pod

spec:

containers:

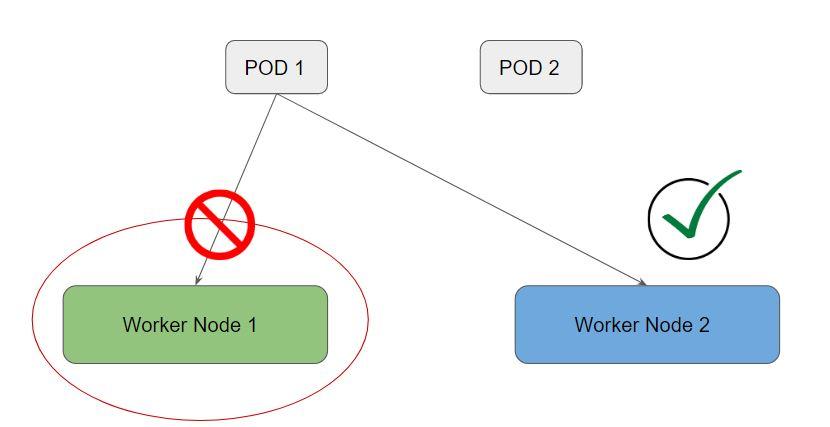
- name: repotech-container

image: nginx

**Taints and Toleration**

Understanding Taints:

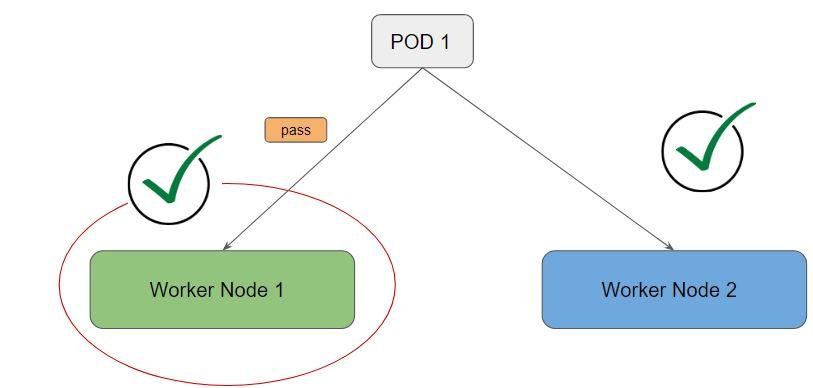
Taints are used to repel the pods from a specific node.



Understanding Toleration:

In order to enter the taint worker node, you need a special pass.

This pass is called toleration.



kubectl taint nodes kubeadm-worker-01 key=value:NoSchedule

kubectl describe node kubeadm-worker-01

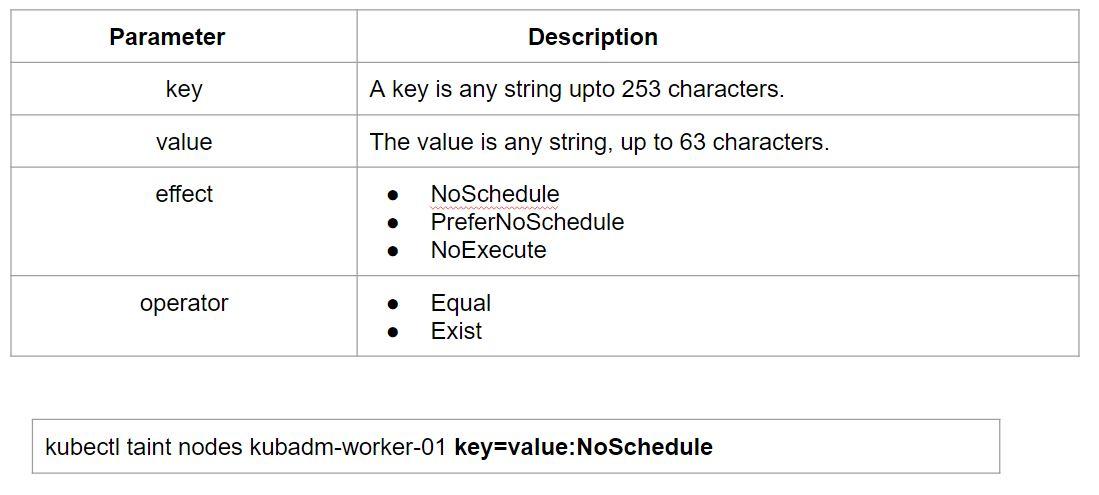
kubectl run nginx --image=nginx

Kubeadm-worker-0 kubectl taint nodes k1 key=value:NoSchedule-

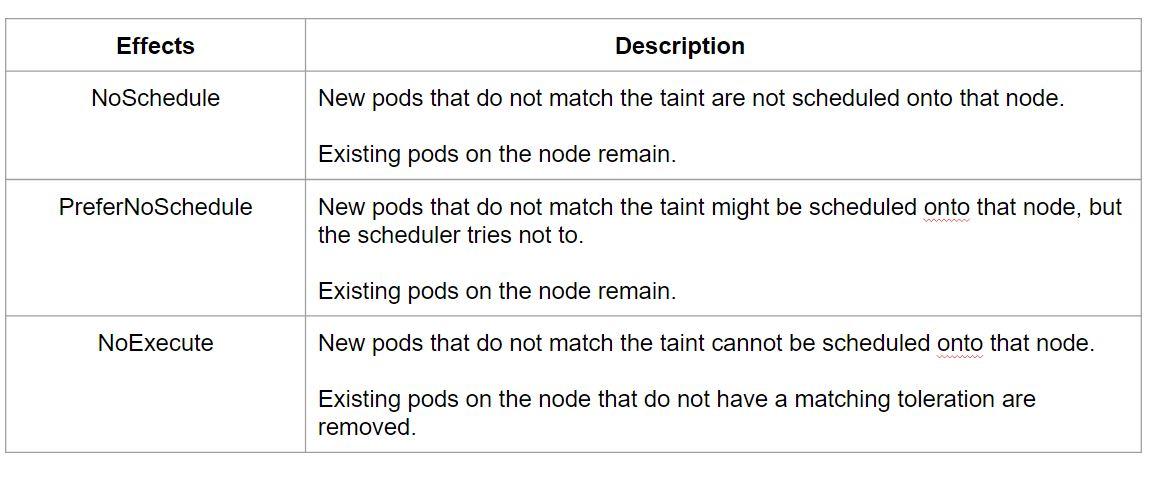
**Components of Taints and Tolerations**

A taint allows a node to refuse pod to be scheduled unless that pod has matching toleration.

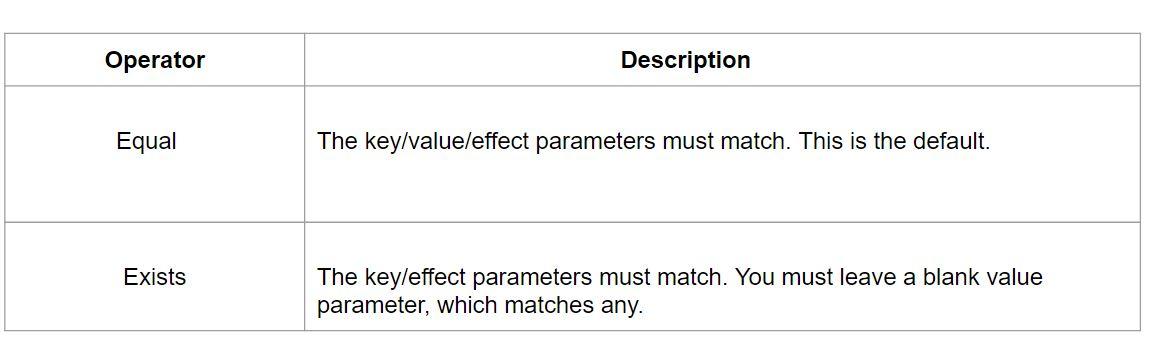
We can apply toleration to a pod within the PodSpec



The following table shows the effects:



The following are the two primary operators.



apiVersion: v1

kind: Pod

metadata:

name: withtolaration

labels:

env: test

spec:

containers:

- name: nginx

image: nginx

imagePullPolicy: IfNotPresent

tolerations:

- key: "key"

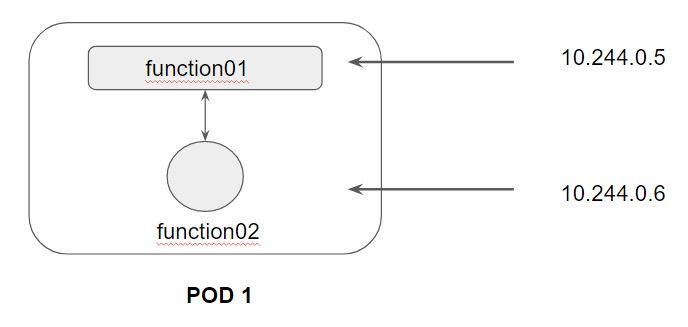
operator: "Equal"

value: "value"

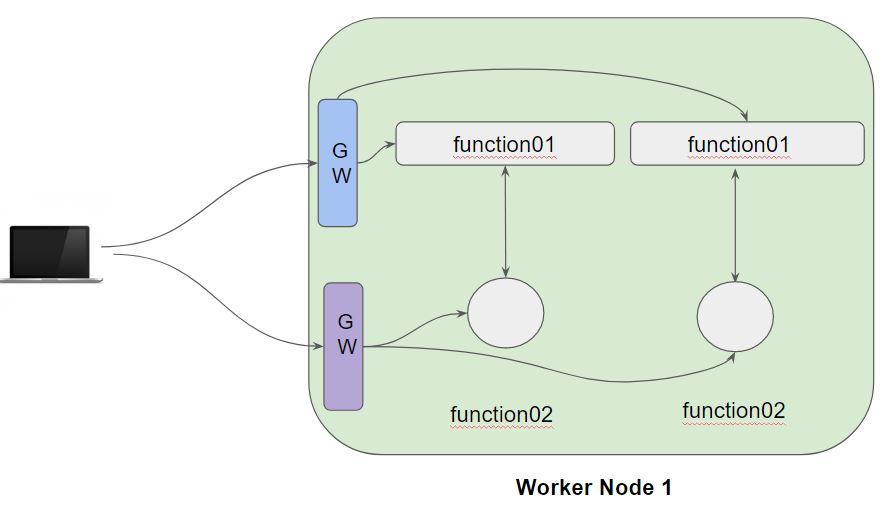
effect: "NoExecute"

**Overview of Service**

Whenever you create a Pod, the containers created will have Private IP addresses.



Following is a high-level diagram on the functionality of Service:



In a Kubernetes cluster, each Pod has an internal IP address.

Pods are generally ephemeral, they can come and go anytime.

We can make use of service which acts as a gateway and can get us connected with right set of pods.

Service is an abstract way of exposing application running in the pods as a network service.

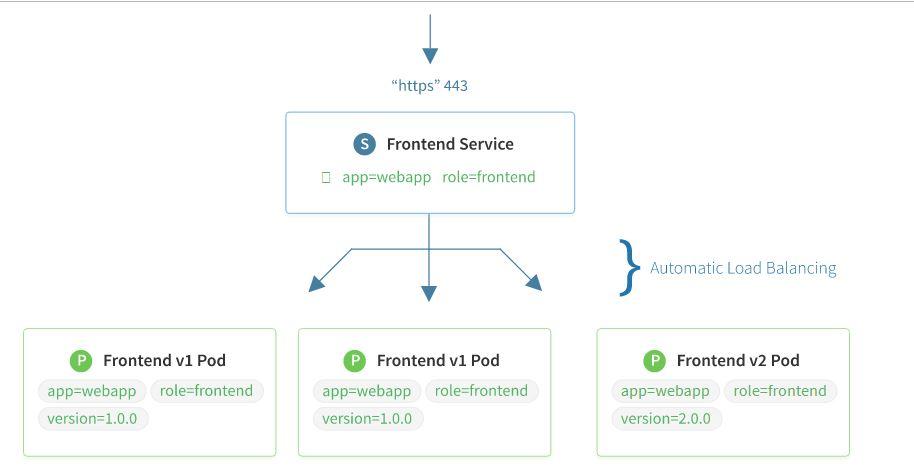
There are several types of Kubernetes Services which are available:

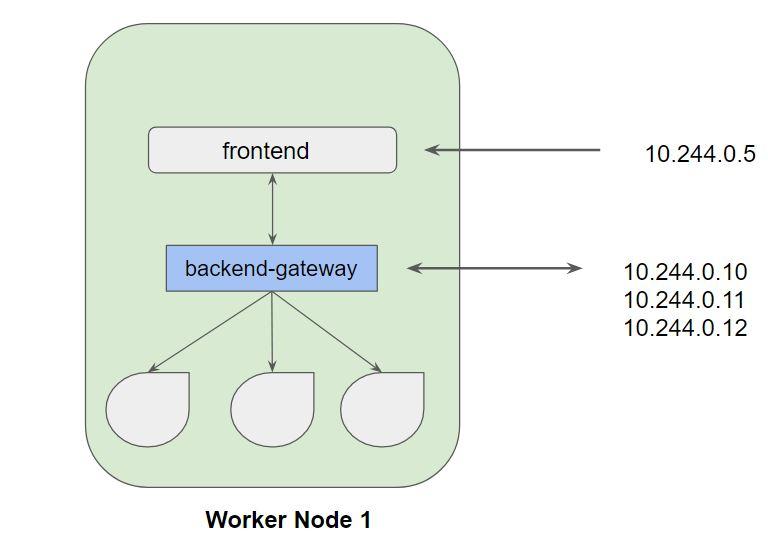
* NodePort
* ClusterIP
* LoadBalancer
* ExternalName

**Creating our first Service and Endpoint**

Kubernetes Service can act as an abstraction which can provide a single IP address and DNS through which pods can be accessed.

Endpoints track the IP address of the objects that service can send traffic to.





Service.yaml:

kind: Service

metadata:

name: repo-service

spec:

ports:

- port: 8080

targetPort: 80

endpoint.yaml

apiVersion: v1

kind: Endpoints

metadata:

name: repo-service

subsets:

- addresses:

- ip: <ipofpod>

ports:

- port: 80

apiVersion: v1

kind: Service

metadata:

name: repo-service1

spec:

selector:

app: nginx

ports:

- port: 8081

targetPort: 80

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 1

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

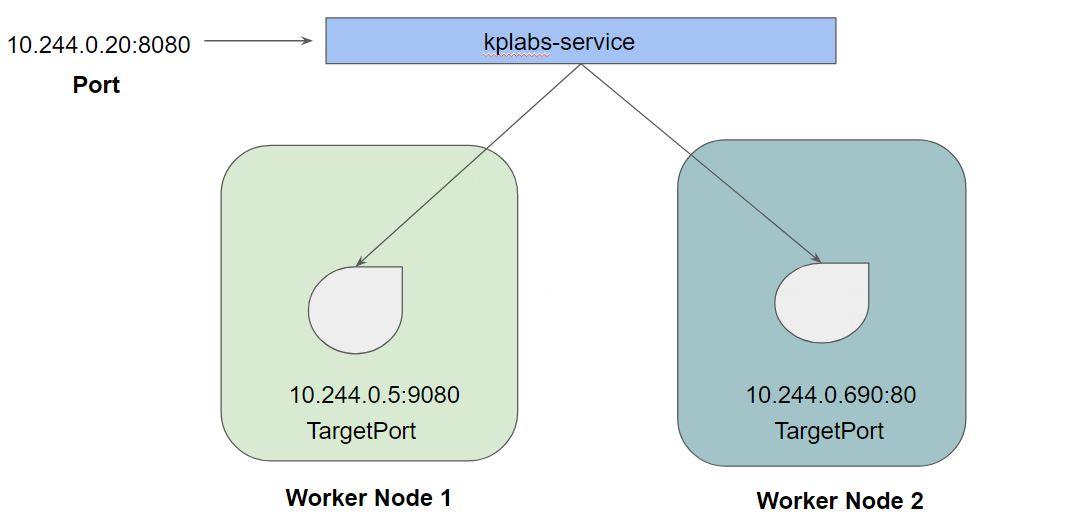
**Service Type - ClusterIP**

Whenever the service type is ClusterIP, an internal cluster IP address is assigned to the service.

Since an internal cluster IP is assigned, it can only be reachable from within the cluster.

This is a default ServiceType.

**Port vs TargePort**

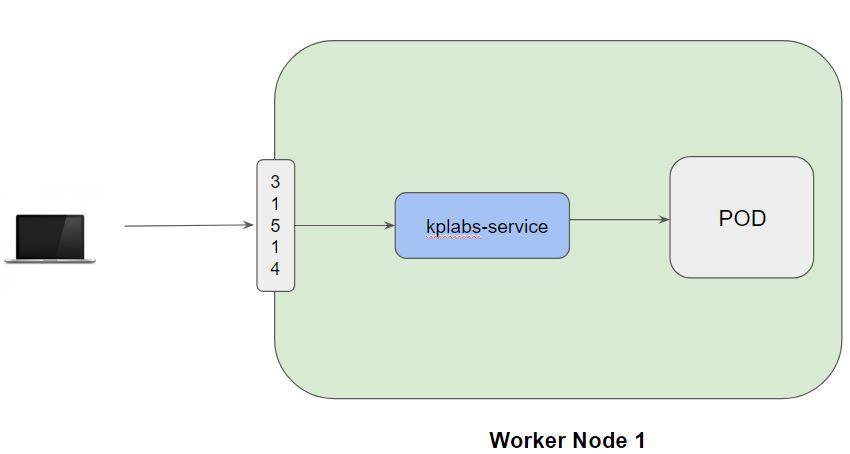
****

**Service Type - NodePort**

From the name, we can identify that it has to do with opening a port on the nodes.

If the service type is NodePort, then Kubernetes will allocate a port (default: 30000-32767) on every worker node.

Each node will proxy that port into your service.



apiVersion: v1

kind: Service

metadata:

name: repo-nodeport

spec:

selector:

run: nginx

type: NodePort

ports:

- port: 80

targetPort: 80

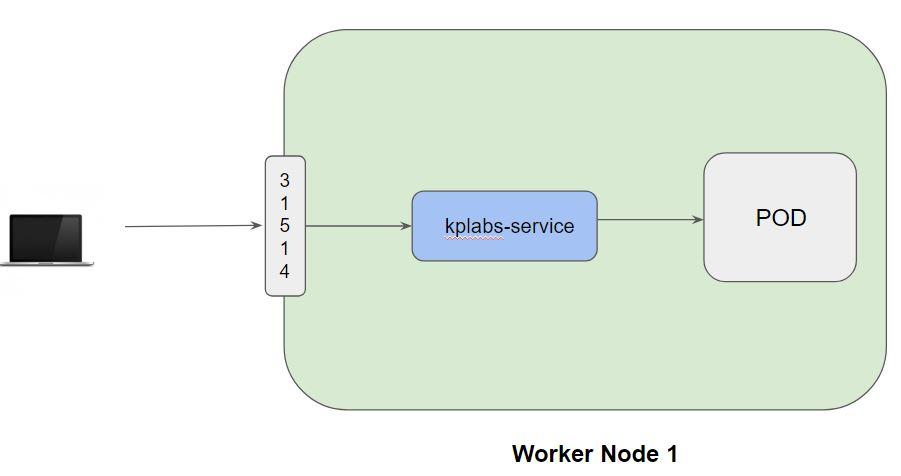
**Service Type - LoadBalancer**

Challenges with NodePort

We know that NodePort ServiceType will assign a node in all the worker node which can forward the traffic to the underlying service.

Challenge in NodePort: We need to access it via IP/DNS:Port

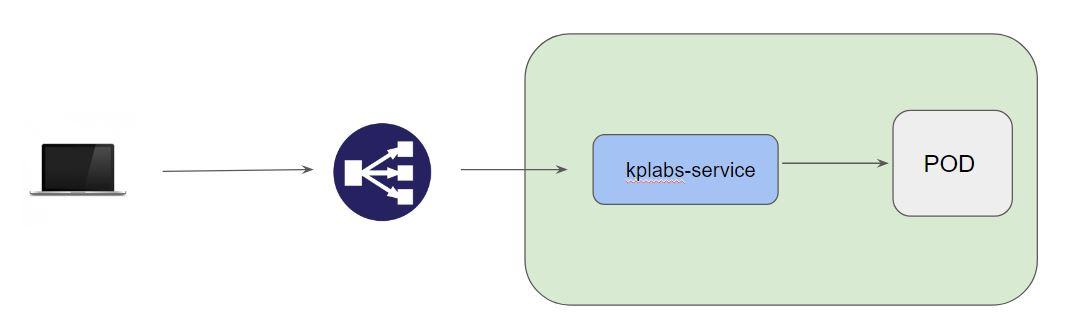
Example: google.com:31514



Understanding LoadBalancer Service Type

LoadBalancer Service Type will automatically deploy an external load balancer.

This load balancer takes care of routing requests to the underlying service.



important Pointers

The overall implementation of LoadBalancer depends on your Cloud Provider.

If you plan to use it in bare-metal, then you will have to provide your own load balancer implementation.

**Create a service - Exposing Pod**

kubectl expose pod nginx --name nginx-svc --port=80 --target-port=80

**Ceate a service - Expose Deployment**

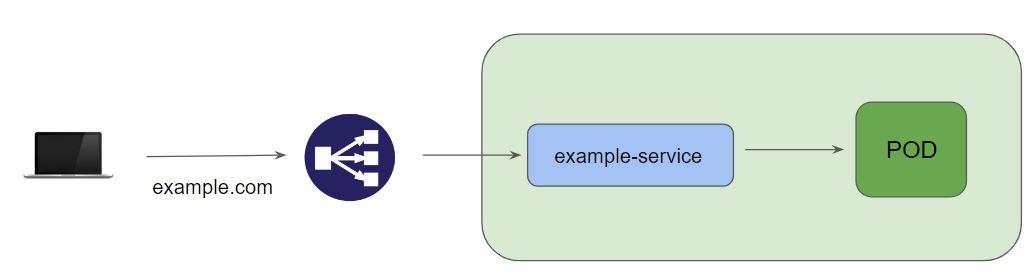
kubectl expose deployment nginx --name nginx-dep-svc --port=80 --target-port=8000

**Create a NodePort Service**

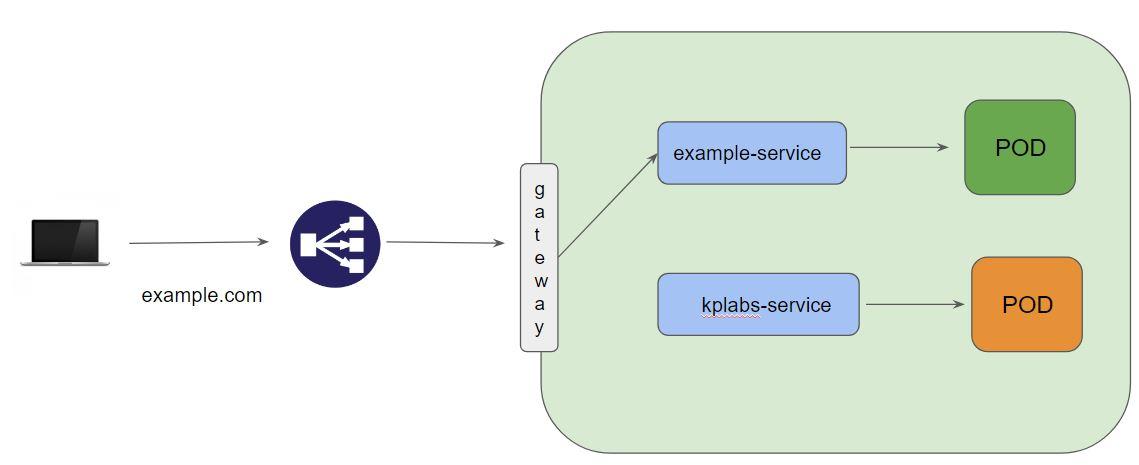
kubectl expose deployment nginx --name nodeport-svc --port=80 --target-port=8000 --type=NodePort

**Ingress**

Whenever making use of the LoadBalancer Service Type, out of the box, you can make use of a single website.



With ingress, we can set up multiple rules through which traffic can be routed to an appropriate service depending on the URL of the website that is requested.



Kubernetes Ingress is a collection of routing rules which governs how external users access the services running within the Kubernetes cluster.

Ingress can provides various features which includes:

* Load Balancing
* SSL Termination
* Named-based virtual hosting

**Ingress Resource and Ingress Controllers**

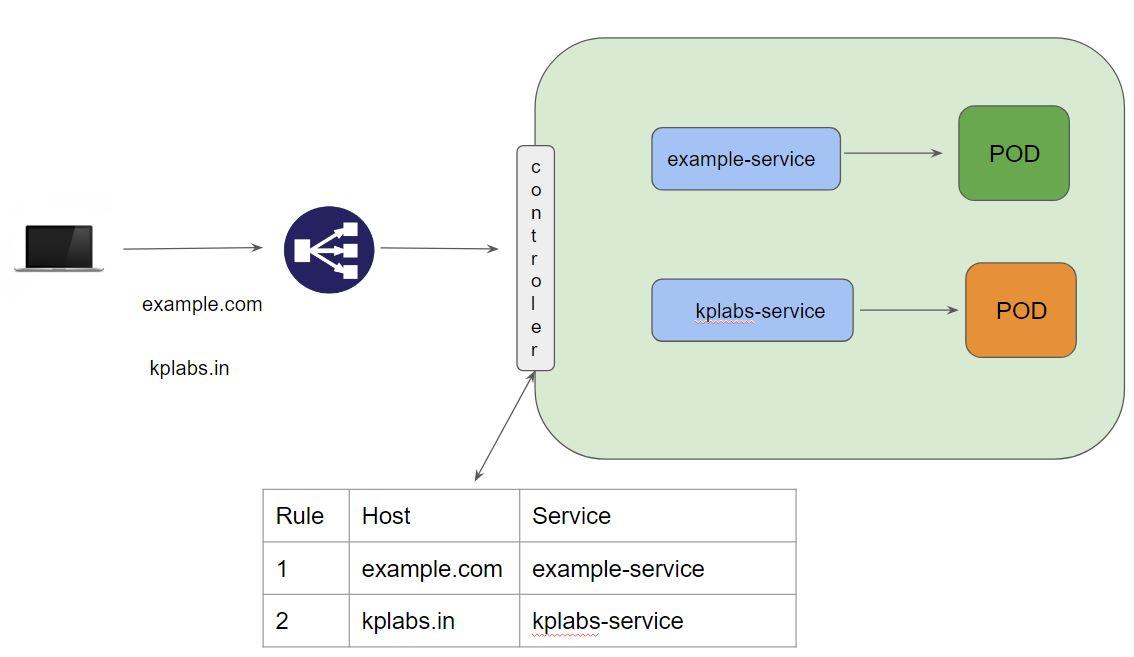
There are two sub-components when we discuss about Ingress:

* Ingress Resource
* Ingress Controllers

Ingress Resource contains set of routing rules based on which traffic is routed to a service.

Ingress Controller takes care of the Layer 7 proxy to implement ingress rules.

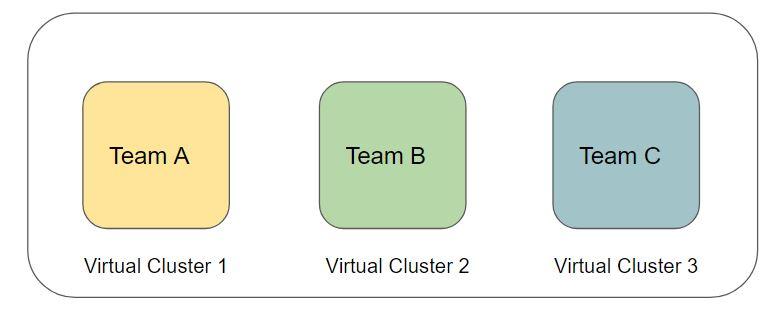
You must have an ingress controller to satisfy an Ingress. Only creating an Ingress resource has no effect



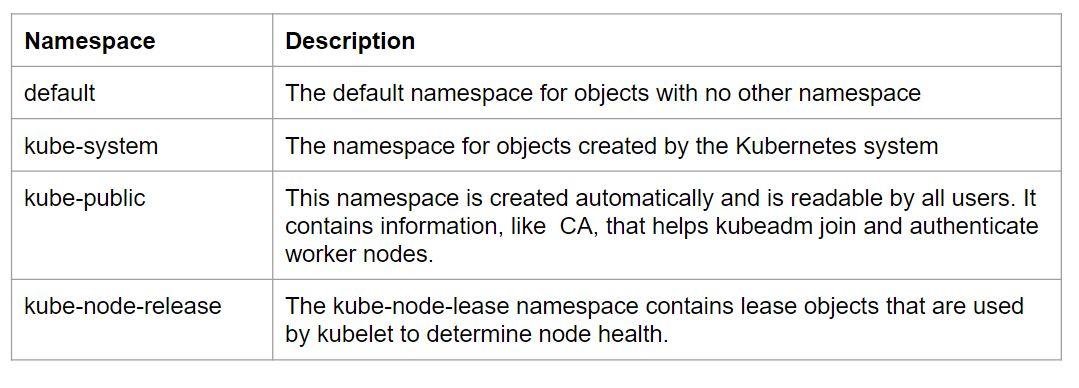
[Ingress | Kubernetes](https://kubernetes.io/docs/concepts/services-networking/ingress/)

**Namespace**

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.



Following is the list of namespaces that are available in Kubernetes:



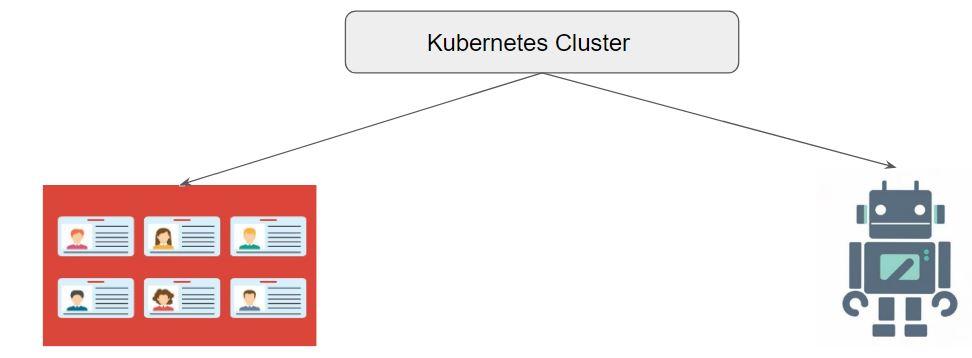
kubectl get pods --all-namespaces to see all objects in all ns

**Service Accounts**

Understanding Authentication

Kubernetes Clusters have two categories of users:

* Normal Users.
* Service Accounts

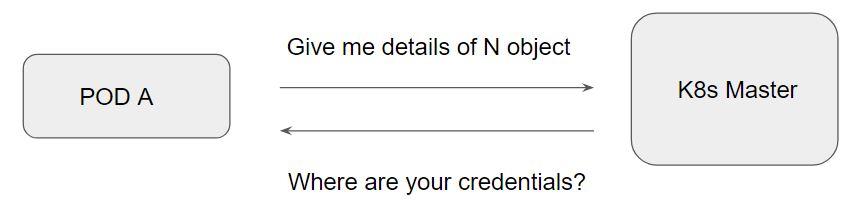


Understanding Service Accounts

Service Accounts allows the Pods to communicate with the API Server

Let’s understand with a use-case:

Application within Pod A wants to retrieve an object within your K8S cluster.



Following diagram indicates the working of service account:



Important Pointer

Service Accounts are namespaced.

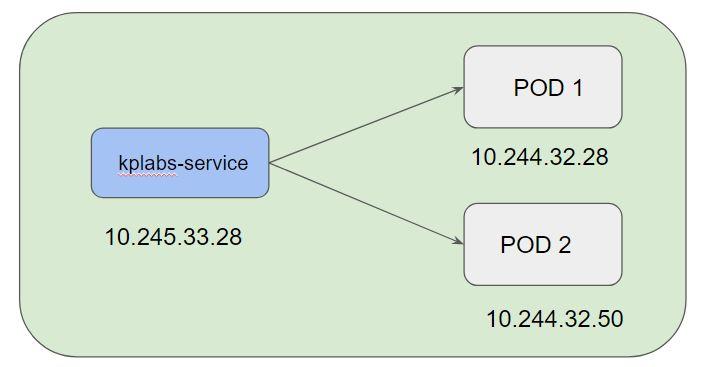
Default service account gets automatically created when you create a namespace

PODS are automatically mounted with the default service accounts.

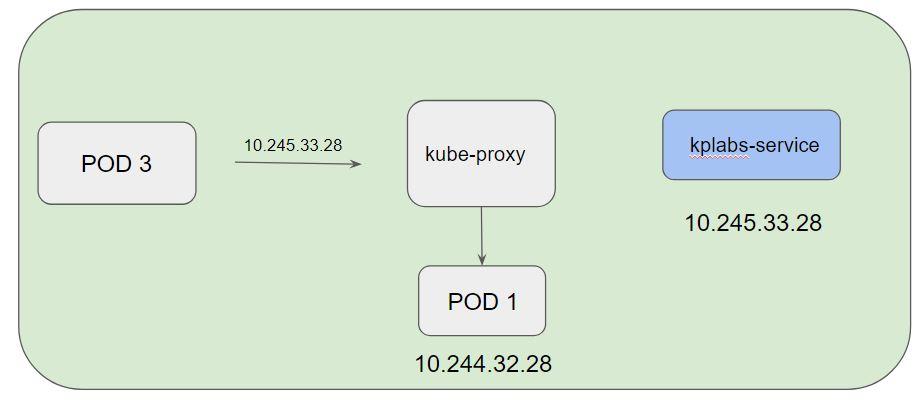
**kube-proxy**

kube-proxy is primarily responsible for forwarding request from ClusterIP to Pod IP.

kube-proxy also takes care of implementing a form of virtual IP for Services



When there is a request for 10.245.33.28, this request is intercepted and redirected to one of the PODS associated with the service.



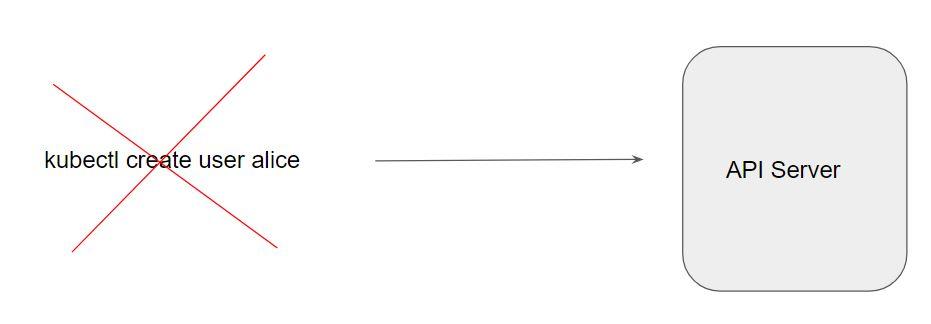
**Authentication**

Kubernetes Clusters have two categories of users:

* Normal Users.
* Service Accounts

Kubernetes does not have objects which represent normal user accounts.

Kubernetes does not manage the user accounts natively.



There are multiple ways in which we can authenticate. Some of these include:

* Usernames / Passwords.
* Client Certificates
* Bearer Tokens

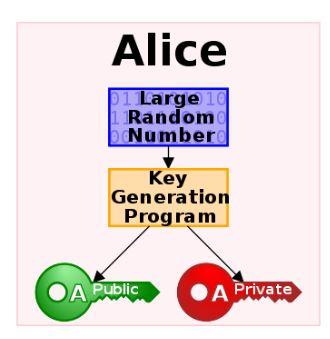
**Asymmetric Key Encryption**

Asymmetric cryptography uses public and private keys to encrypt and decrypt data.

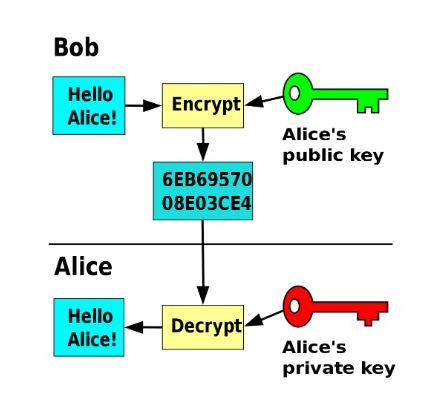
One key in the pair can be shared with everyone; it is called the public key. The other key in the pair is kept secret; it is called the private key.

Either of the keys can be used to encrypt a message; the opposite key from the one used to encrypt the message is used for decryption.

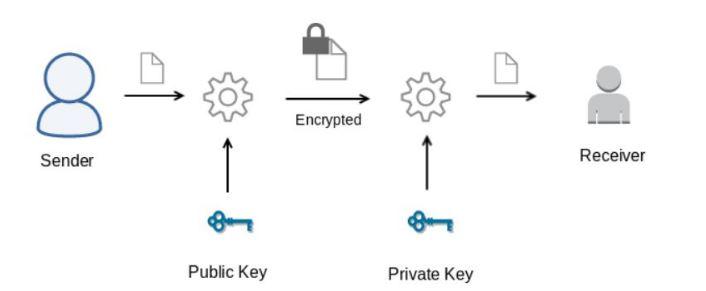
Step 1: Generation of Keys



Step 2: Encryption and Decryption



High-Level Steps:

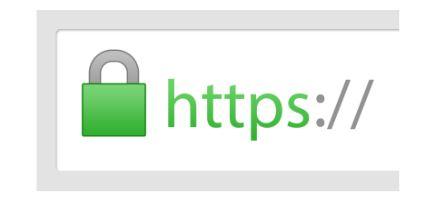


**Understanding SSL/TLS**

HTTPS is an extension of HTTP.

In HTTPS, the communication is encrypted using Transport Layer Security (TLS)

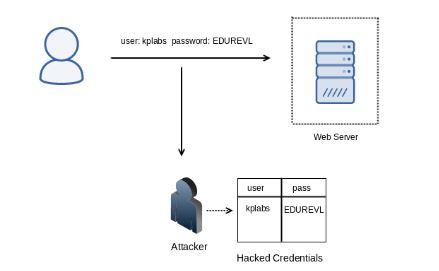
The protocol is therefore also often referred to as HTTPS over TLS or HTTP over SSL.



Scenario 1: MITM Attacks

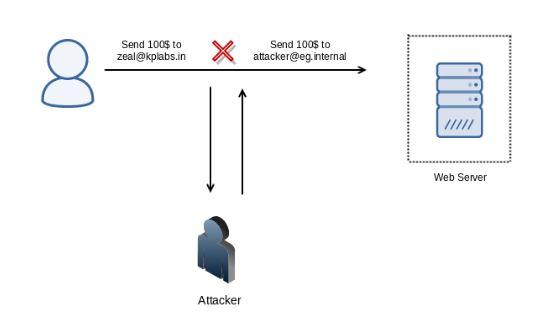
User is sending their username and password in plaintext to a Web Server for authentication over a network.

There is an Attacker sitting between them doing a MITM attack and storing all the credentials he finds over the network to a file:



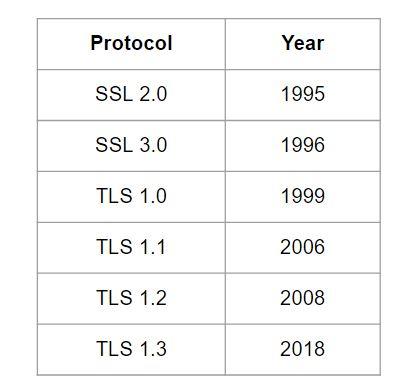
Scenario 2: MITM & Integrity Attacks

Attacker changing the payment details while packets are in transit.



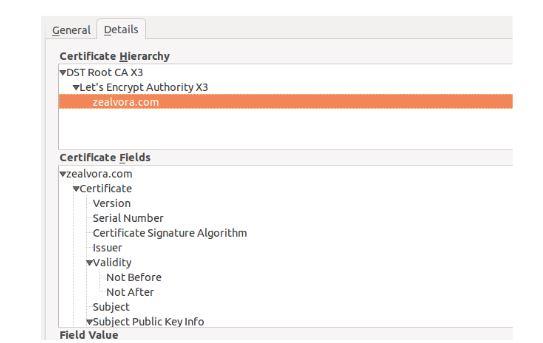
Introduction to SSL/TLS

To avoid the previous two scenarios (and many more), various cryptographic standards were clubbed together to establish secure communication over an untrusted network and they were known as SSL/TLS.



Every website has a certificate (like a passport that is issued by a trusted entity).

The certificate has a lot of details like the domain name it is valid for, the public key, validity, and others.



The browser (clients) verifies if it trusts the certificate issuer.

It will verify all the details of the certificate.

It will take the public key and initiate a negotiation.

Asymmetric key encryption is used to generate a new temporary

Symmetric key which will be used for secure communication.

**Creating TLS Certificate for Authentication**

Our goal is to set up authentication based on X509 Client certificates.

User A should be able to authenticate with Kubernetes Cluster with his certificates.



Pre-Requisite:

- Create a Linux container / Linux server were we can run openssl commands.

yum -y install openssl nano

- Connect the linux box with existing K8s setup.

Kubectl Installation:

curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl

Step 1: Create a new private key and CSR

openssl genrsa -out repo.key 2048

openssl req -new -key repo.key -out repo.csr -subj "/CN=repo/O=repotech"

Step 2: Encode the csr

cat repo.csr | base64 | tr -d '\n'

Step 3: Generate the Kubernetes Signing Request

Use this config format if you are using Kubernetes Version Earlier then 1.22

apiVersion: certificates.k8s.io/v1beta1

kind: CertificateSigningRequest

metadata:

name: repo-csr

spec:

groups:

- system:authenticated

request:

usages:

- digital signature

- key encipherment

- client auth

Use this config format if you are using Kubernetes Version Above 1.22

apiVersion: certificates.k8s.io/v1

kind: CertificateSigningRequest

metadata:

name: repo-csr

spec:

signerName: kubernetes.io/kube-apiserver-client

groups:

- system:authenticated

request: <put your certificate>

usages:

- digital signature

- key encipherment

- client auth

Step 4: Apply the Signing Requests:

kubectl apply -f signingrequest.yaml

Step 5: Approve the csr

kubectl certificate approve repo-csr

Step 6: Download the Certificate from csr

kubectl get csr repo-csr -o jsonpath='{.status.certificate}' | base64 -d > repo.crt

Step 7: Create a new context

kubectl config set-credentials repo --client-certificate=repo.crt --client-key=repo.key

Step 8: Set new Context

kubectl config set-context repo-context --cluster do-blr1-kplabs-k8s --user=repo

Step 9: Use Context to Verify

kubectl --context=repo-context get pods

**Authorization**

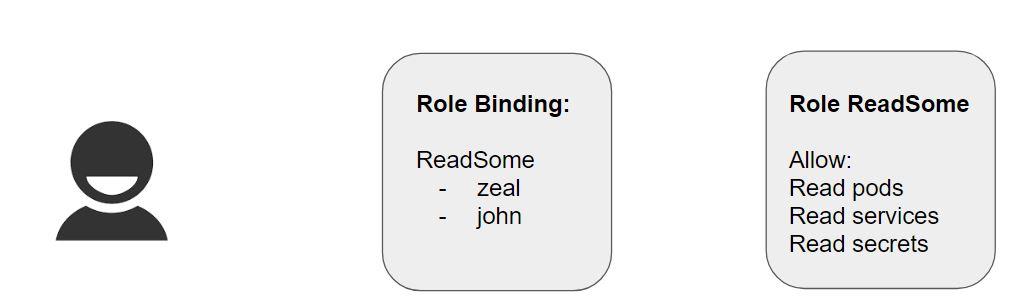
By default, the user does not have any permission granted to them.

Kubernetes provides multiple authorization modules like:

* AlwaysAllow
* AlwaysDeny
* Attribute-Based Access Control (ABAC)
* **Role-based access control (RBAC)**
* Node
* WebHook

A role contains rules that represent a set of permissions

A role binding grants the permissions defined in a role to a user or set of users.



**API Groups, Resources and Verbs**

Overview of API Groups

API Groups makes it easier to extend the Kubernetes API.

Currently there are several groups available:

The core group, often referred to as the legacy group, is at the REST path /api/v1

The named groups are at REST path /apis/$GROUP\_NAME/$VERSION

We have flexibility to enable/disable a specific API Groups in API server.

Resources and Verbs

Within a specific API Group, there are multiple resources available.

Example: Pods, namespaces, nodes,

Verbs are generally the type of requests that can be sent to a resource.

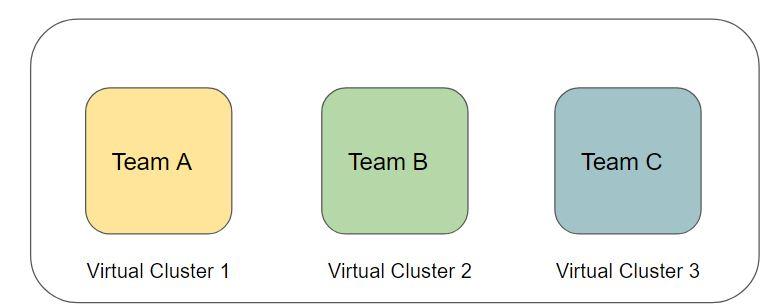
Example Verbs:

Get, list, create, update, patch, watch and others.

**Access Control based on Namespace**

Example Use-Case:

You provide all the users within Team A full access on “Pods” resources [only for Team A NS]



Important Pointer:

A Role can only be used to grant access to resources within a single namespace.

There can be multiple role with the same name in two different namespaces.

**ClusterRole and ClusterRoleBinding**

A Role can only be used to grant access to resources within a single namespace.

A ClusterRole can be used to grant the same permissions as a Role, but because they are cluster-scoped, they can also be used to grant access to:

cluster-scoped resources (like nodes)

namespaced resources (like pods) across all namespaces

Important Pointers:

A RoleBinding may also reference a ClusterRole to grant the permissions to namespaced resources defined in the ClusterRole within the RoleBinding’s namespace

This allows administrator to have set of central policy which can be attached via RoleBinding so it is applicable at a per-namespace level.

**Network Policies**

NetworkPolicies apply to a connection with a pod on one or both ends, and are not relevant to other connections.

The entities that a Pod can communicate with are identified through a combination of the following 3 identifiers:

1. Other pods that are allowed (exception: a pod cannot block access to itself)
2. Namespaces that are allowed
3. IP blocks (exception: traffic to and from the node where a Pod is running is always allowed, regardless of the IP address of the Pod or the node)

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: default-deny-pod

namespace: default

spec:

podSelector:

matchLabels:

run: pod01

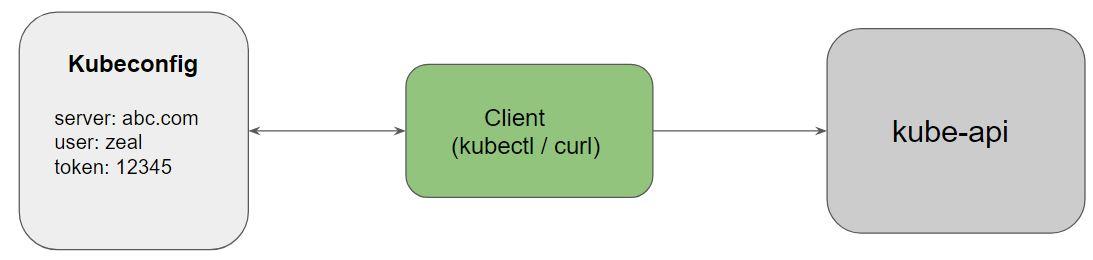
policyTypes:

- Ingress

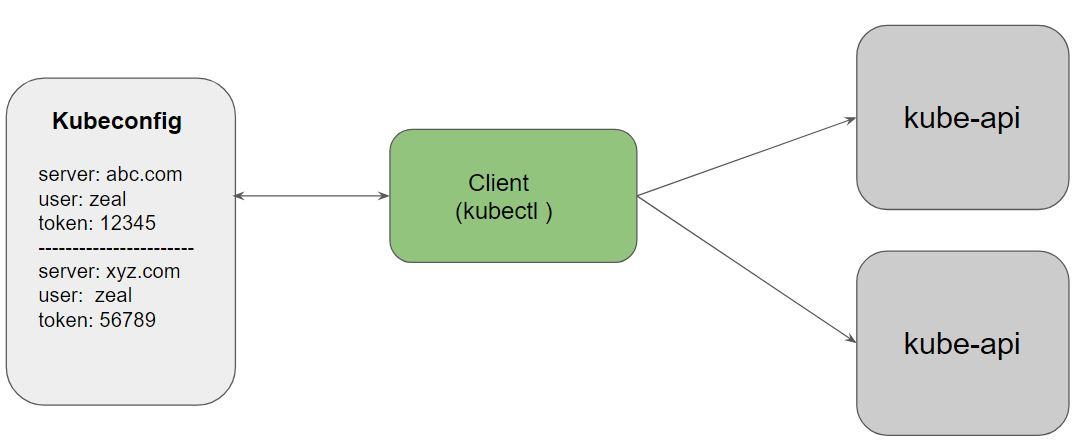
- Egress

**Understanding Kubeconfig**

Kubectl command uses kubeconfig files to find the information about the cluster, username, authentication mechanisms and others.

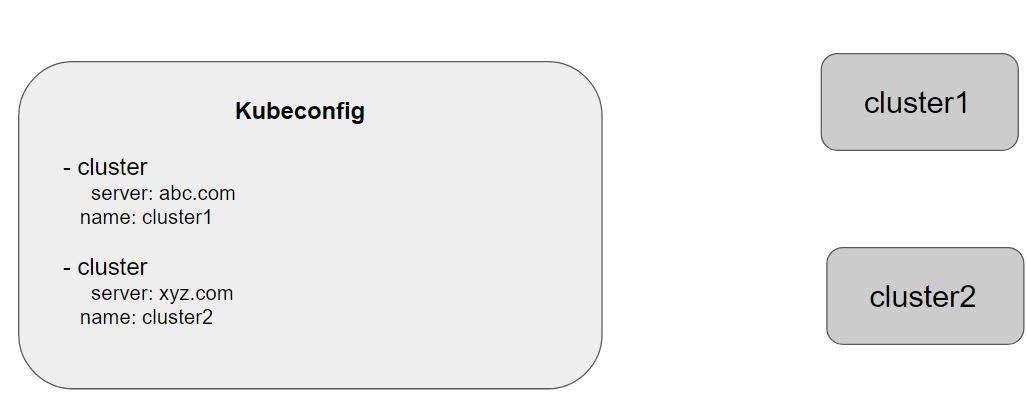


Kubeconfig file can also have details associated with multiple kubernetes clusters.



Step 1: Understanding Clusters Field

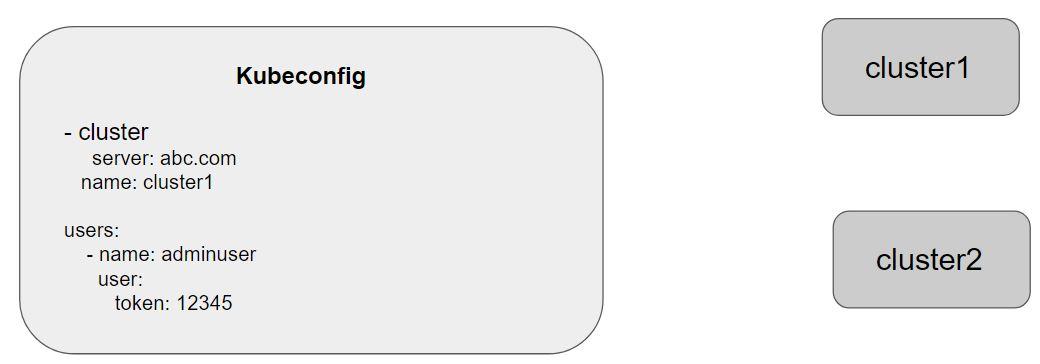
Cluster field has details related to the URL of your Kubernetes Cluster and it’s associated information.



Step 2: Users Field

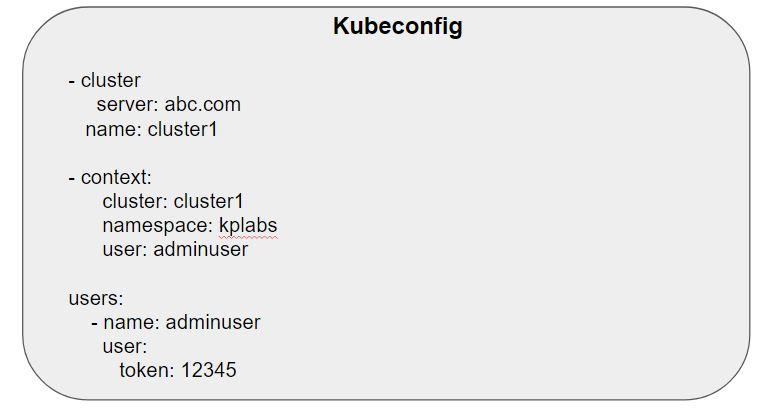
User field contains authentication specific information like username, passwords.

There can be different type of authentication mechanisms (user/pass, certificates, tokens etc



Step 3: Context Field

Context groups information related to cluster, user and namespace.



Create KubeConfig from scratch

1. Add cluster details.

kubectl config --kubeconfig=base-config set-cluster development --server=https://1.2.3.4

2. Add user details

kubectl config --kubeconfig=base-config set-credentials experimenter --username=dev --password=some-password

3. Setting Contexts

kubectl config --kubeconfig=base-config set-context dev-frontend --cluster=development --namespace=frontend --user=experimenter

4. Repeating above steps for second cluster

kubectl config --kubeconfig=base-config set-cluster production --server=https://4.5.6.7

kubectl config --kubeconfig=base-config set-context prod-frontend --cluster=production --namespace=frontend --user=experimenter

Next Steps:

1. View Kubeconfig

kubectl config --kubeconfig=base-config view

2. Get current conext information:

kubectl config --kubeconfig=base-config get-contexts

3. Switch Conexts:

kubectl config --kubeconfig=base-config use-context dev-frontend

**Kubernetes Secrets**

aws secret manager

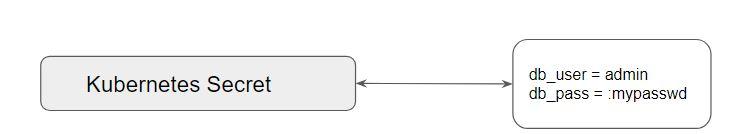
aws system manager parameter store

hashicorp vault

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key.

Allows customers to store secrets centrally to reduce risk of exposure.

Stored in ETCD database.



Following is the syntax for creating a secret via CLI:

kubectl create secret [TYPE] [NAME] [DATA]

Elaborating Type:

i) Generic:

File (--from-file)

directory

literal value

ii) Docker Registry

iii) TLS

Generating Secret Based on Generic - Literal Value

kubectl create secret generic firstsecret --from-literal=dbpassword=mypassword123

Generating Secret Based on Generic - File

kubectl create secret generic secondsecret --from-file=./credentials.txt

To View Secret from CLI

kubectl get secret firstsecret -o yaml

##### **secret-data.yaml**

apiVersion: v1

kind: Secret

metadata:

name: thirdsecret

type: Opaque

data:

username: ZGJhZG1pbg==

password: bXlwYXNzd29yZDEyMw==

##### **secret-stringdata.yaml**

apiVersion: v1

kind: Secret

metadata:

name: stringdata

type: Opaque

stringData:

config.yaml: |-

username: dbadmin

password: mypassword

##### **secretpod.yaml**

apiVersion: v1

kind: Pod

metadata:

name: secretmount

spec:

containers:

- name: secretmount

image: nginx

volumeMounts:

- name: foo

mountPath: "/etc/foo"

readOnly: true

volumes:

- name: foo

secret:

secretName: firstsecret

##### **Mounting Secrets as Environment Variables**

File Name: secret-env.yaml

apiVersion: v1

kind: Pod

metadata:

name: secret-env

spec:

containers:

- name: secret-env

image: nginx

env:

- name: SECRET\_USERNAME

valueFrom:

secretKeyRef:

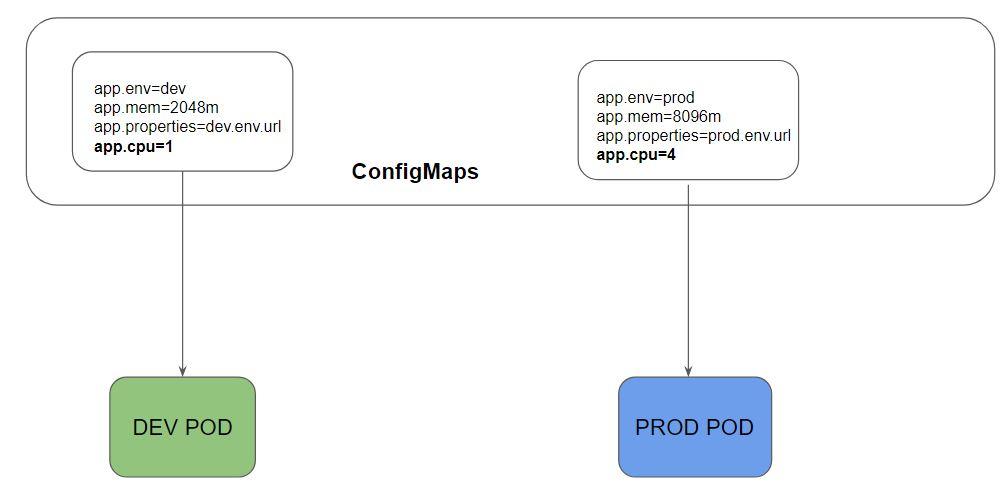
name: firstsecret

key: dbpass

restartPolicy: Never

## **ConfigMaps**

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable.



app.env=dev

app.mem=2048m

app.properties=dev.env.url

apiVersion: v1

kind: Pod

metadata:

name: configmap-pod

spec:

containers:

- name: test-container

image: nginx

volumeMounts:

- name: config-volume

mountPath: /etc/config

volumes:

- name: config-volume

configMap:

name: dev-properties

restartPolicy: Never

## **Overview of Docker Volumes**

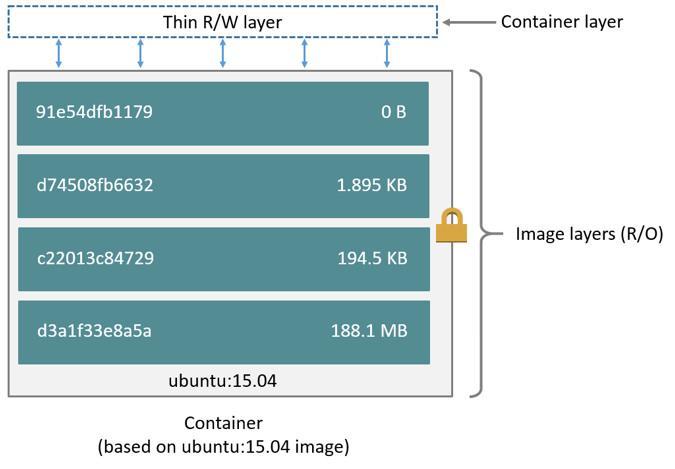
Challenges with files in Container Writable Layer

By default, all files created inside a container are stored on a writable container layer. This means that:

The data doesn’t persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.

Writing into a container’s writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel.

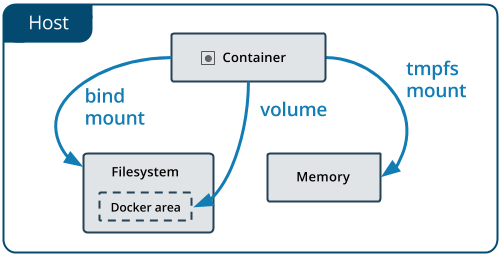
This extra abstraction reduces performance as compared to using data volumes, which write directly to the host filesystem.



Ideal Approach for Persistent Data

Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops: volumes, and bind mounts.

If you’re running Docker on Linux you can also use a tmpfs mount.



Important Pointers to Remember:

A given volume can be mounted into multiple containers simultaneously.

When no running container is using a volume, the volume is still available to Docker and is not removed automatically.

When you mount a volume, it may be named or anonymous. Anonymous volumes are not given an explicit name when they are first mounted into a container, so Docker gives them a random name that is guaranteed to be unique within a given Docker host.

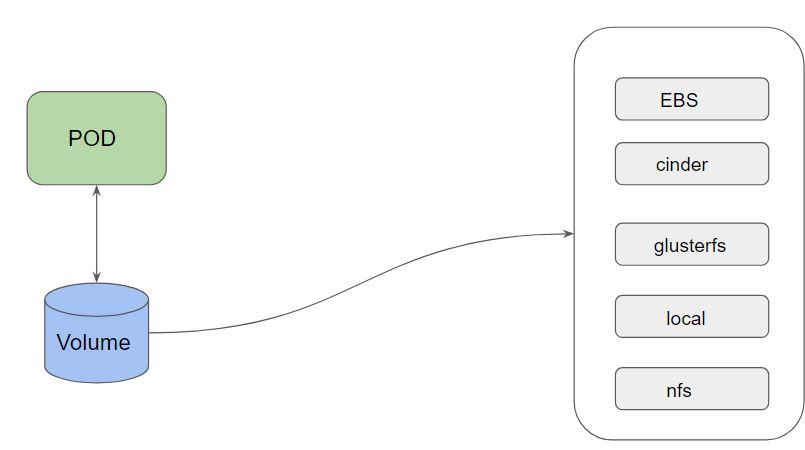
## 

## **Volume in Kubernetes**

On-disk files in a Container are ephemeral.

When there are multiple containers that want to share the same data, it becomes a challenge.

One of the benefits of Kubernetes is that it supports multiple types of volumes.



## **PersistentVolume and PersistentVolumeClaim**

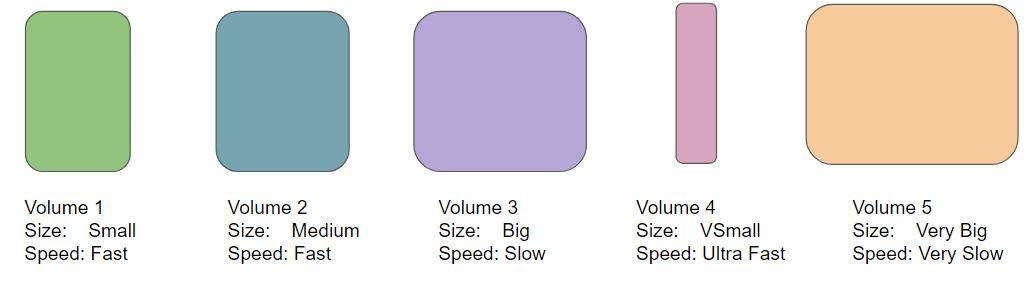
PersistentVolume (PV)

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes

re

Every Volume which is created can be of a different types.

This can be taken care of by the Storage Administrator / Ops Team



apiVersion: v1

kind: PersistentVolume

metadata:

name: block-pv

spec:

storageClassName: manual

capacity:

storage: 10Gi

accessModes:

- ReadWriteOnce

hostPath:

path: /tmp/data

PersistentVolumeClaim (PVC)

A PersistentVolumeClaim is a request for the storage by a user.

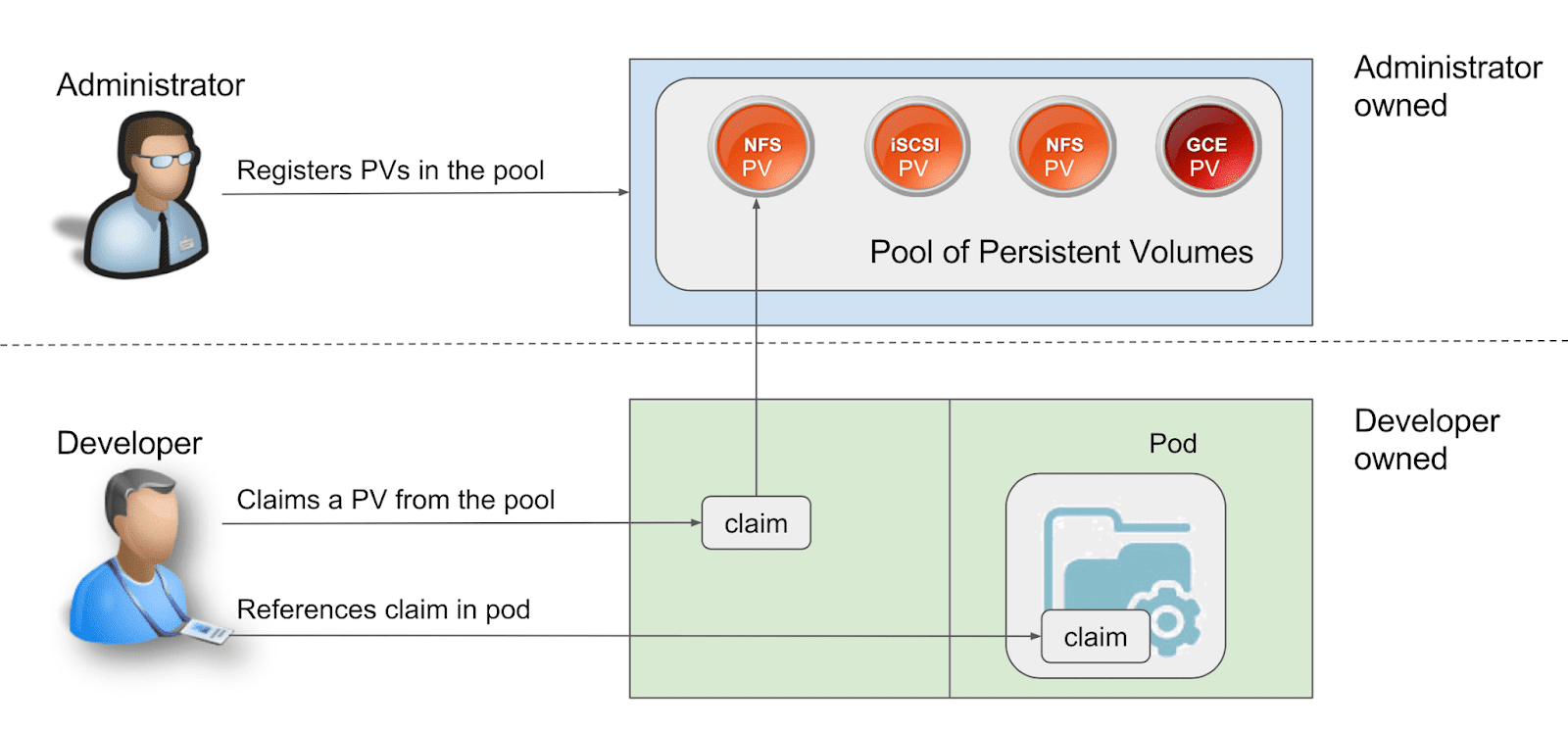
Within the claim, the user needs to specify the size of the volume along with access mode.

Developer:

I want a volume of size 10 GB which has a speed of Fast for my pod.

High-Level Working Steps:

* Storage Administrator takes care of creating PV.
* Developer can raise a “Claim” (I want a specific type of PV).
* Reference that claim within the PodSpec file.



apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: myfirstpvc

spec:

storageClassName: manual

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 10Gi

kind: Pod

apiVersion: v1

metadata:

name: repo-pvc

spec:

containers:

- name: my-frontend

image: nginx

volumeMounts:

- mountPath: "/data"

name: my-volume

volumes:

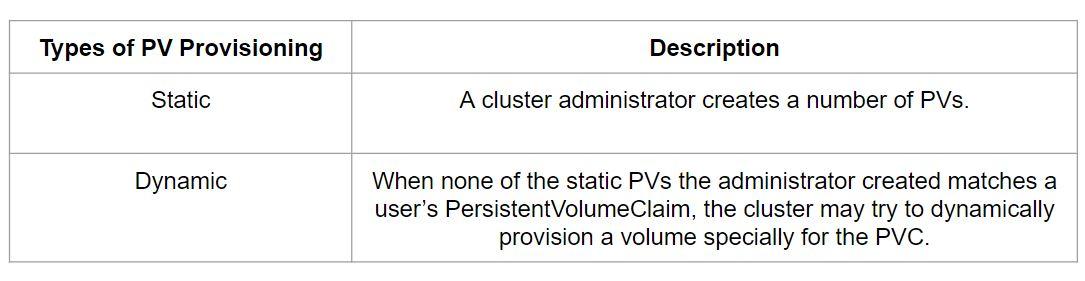
- name: my-volume

persistentVolumeClaim:

claimName: pvc

## **Static vs Dynamic Provisioning of PV**

There are two ways PVs may be provisioned: statically or dynamically.



apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pvcdyn

spec:

storageClassName: local-storage

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 10Gi

## **Storage class**

A StorageClass provides a way for administrators to describe the "classes" of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. Kubernetes itself is unopinionated about what classes represent. This concept is sometimes called "profiles" in other storage systems.

[Storage Classes | Kubernetes](https://kubernetes.io/docs/concepts/storage/storage-classes/)

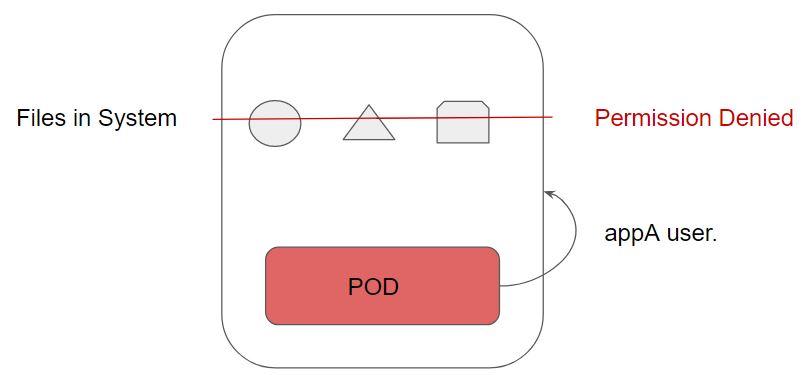
## **Security Contexts**

When you run a container, it runs with the UID 0 (Administrative Privilege)

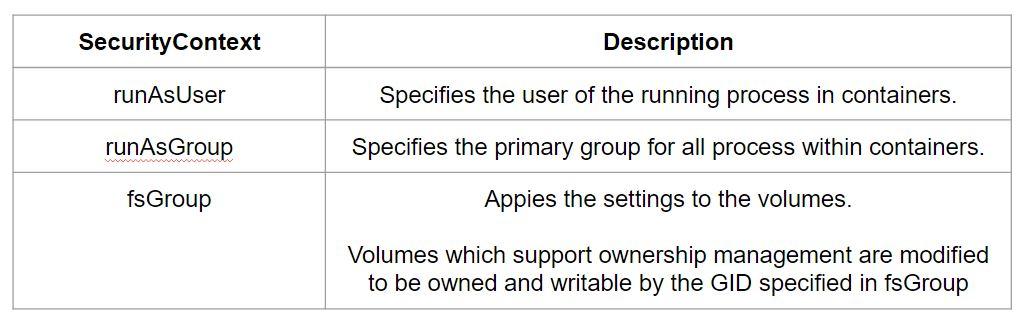
In-case of container breakouts, attacker can get root privileges to your entire system.



We can run POD and container with limited privilege user instead of the ROOT user.



Following are the three important permissions:



apiVersion: v1

kind: Pod

metadata:

name: security-context-demo

spec:

securityContext:

runAsUser: 1000

runAsGroup: 3000

fsGroup: 2000

volumes:

- name: sec-ctx-vol

emptyDir: {}

containers:

- name: sec-ctx-demo

image: busybox:1.28

command: [ "sh", "-c", "sleep 1h" ]

volumeMounts:

- name: sec-ctx-vol

mountPath: /data/demo

securityContext:

allowPrivilegeEscalation: false

---

apiVersion: v1

kind: Pod

metadata:

name: pod-context

spec:

securityContext:

runAsUser: 1000

runAsGroup: 3000

containers:

- name: sec-ctx-demo

image: busybox

command: [ "sh", "-c", "sleep 1h" ]

## **StatefulSets**

StatefulSet is the workload API object used to manage stateful applications.

Manages the deployment and scaling of a set of [Pods](https://kubernetes.io/docs/concepts/workloads/pods/), *and provides guarantees about the ordering and uniqueness* of these Pods.

StatefulSets are valuable for applications that require one or more of the following.

* Stable, unique network identifiers.
* Stable, persistent storage.
* Ordered, graceful deployment and scaling.
* Ordered, automated rolling updates.

apiVersion: v1

kind: Service

metadata:

name: nginx

labels:

app: nginx

spec:

ports:

- port: 80

name: web

clusterIP: None

selector:

app: nginx

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

serviceName: "nginx"

replicas: 2

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: k8s.gcr.io/nginx-slim:0.8

ports:

- containerPort: 80

name: web

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumes:

- name: www

persistentVolumeClaim:

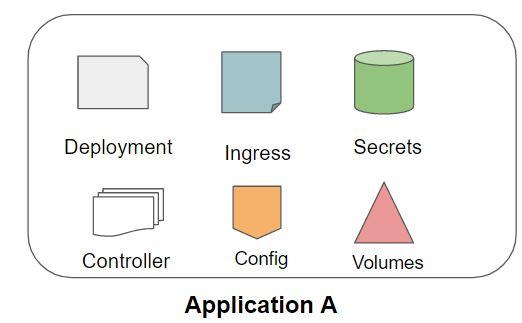
claimName: myfirstpvc

**Helm**

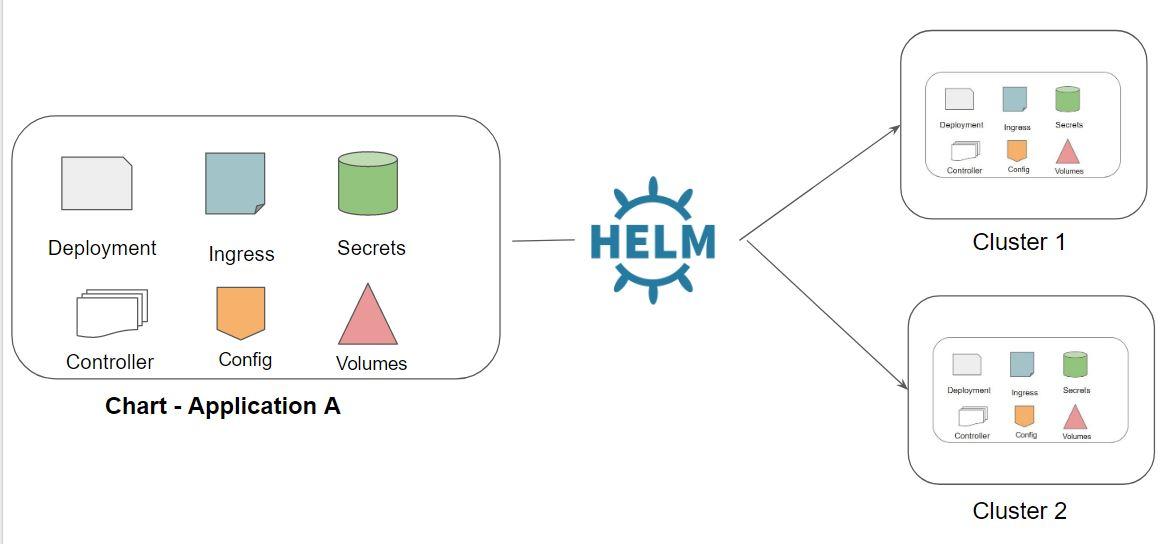
Helm is one of the package manager for Kubernetes.

Kubernetes application can contain lot of lot of objects like:

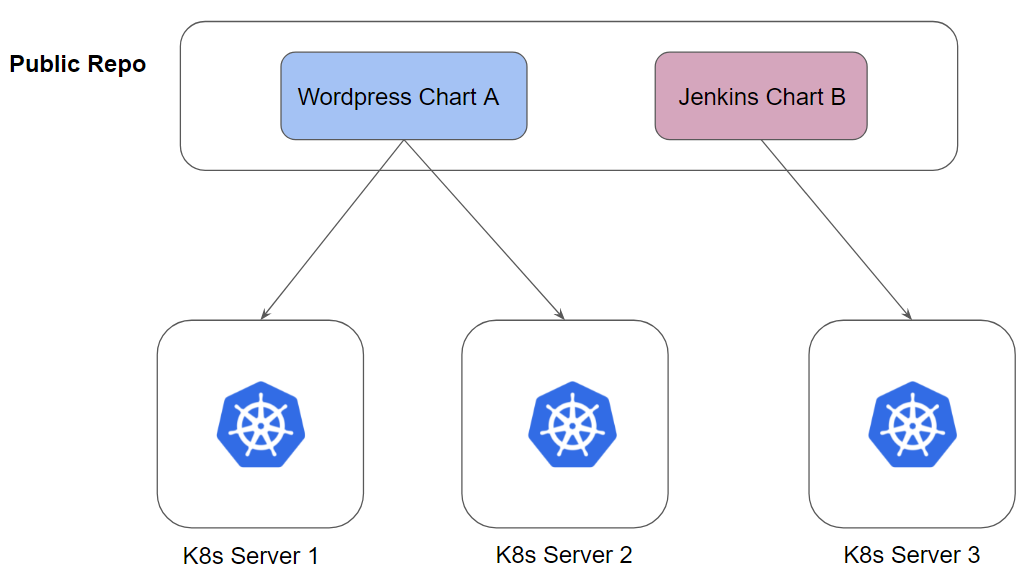
Deployments, Secrets, LoadBalancers, Volumes, services, ingress and others.



Following diagram depicts Helm Charts



All the Helm charts are stored in a central repository through which charts can be pulled and deployd.



Before we conclude, let us revise the concepts:

* A Chart is a Helm package.
* It contains all of the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster.
* A Repository is the place where charts can be collected and shared

Install helm using below link

[Helm | Installing Helm](https://helm.sh/docs/intro/install/)

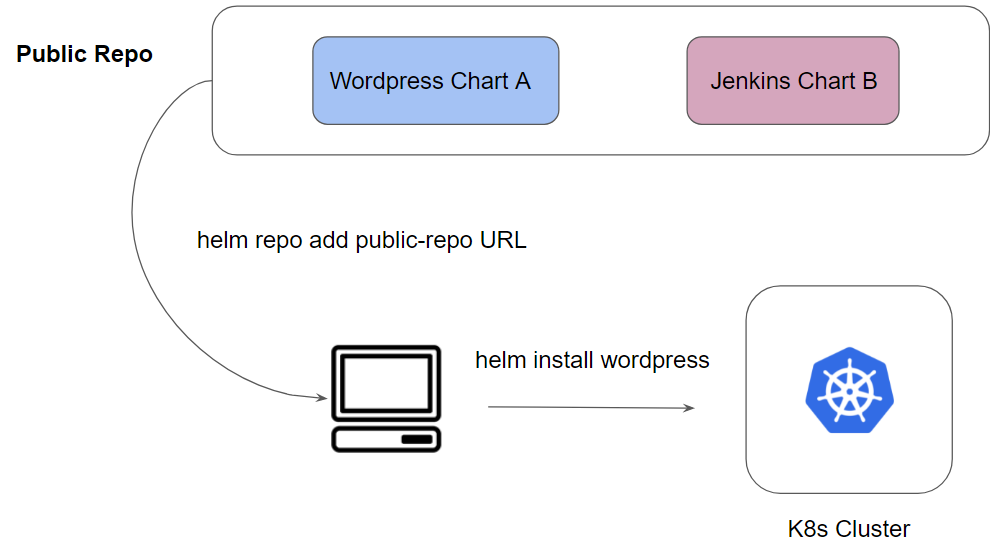
**Deploying Helm Charts**

To install a new package, use the **helm install** command



Helm packages can be placed in local disk or can also be stored centrally in public or private repositories.

Depending on the location of the package, there is a slight change in the installation step.



Important Note:

* Every application packaged in Helm has its own set of requirements. Make sure to read the instructions carefully.
* Install Helm Package only from trusted repositories.

Following table describes all the basic Helm commands:



**Kubernetes Dashboard Creation.**

**RKE:**

https://github.com/rancher/rke/releases/

<https://github.com/justmeandopensource/kubernetes/tree/master/rancher/rke>

<https://youtu.be/qQiKbE9pmEE>

**EKS: (AWS Managed Service)**

Prerequisites:

1. Jump box/GIT bash.
2. Aws cli [Installing or updating the latest version of the AWS CLI - AWS Command Line Interface (amazon.com)](https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html)
3. Eksctl [Installing eksctl - Amazon EKS](https://docs.aws.amazon.com/eks/latest/userguide/eksctl.html)

eksctl create cluster firsteks --region ap-south-1 --nodegroup-name myclusterng --node-type t3.small --nodes 2 --managed (with server)

eksctl create cluster --name my-cluster --region ap-south-1 --fargate (server less)

**kops:**

<https://kubernetes.io/docs/setup/production-environment/tools/kops/>

**GKE cluster create:**