Table of Contents

[What is Containerization ? 2](#_Toc176009907)

[Which applications are the best candidates for containerization? 2](#_Toc176009908)

[What is Kubernetes ? 2](#_Toc176009909)

[Features of Kubernetes 3](#_Toc176009910)

[Kubernetes Architecture: 4](#_Toc176009911)

[Master node 5](#_Toc176009912)

[API server 6](#_Toc176009913)

[**Scheduler** 6](#_Toc176009914)

[**Controller manager** 6](#_Toc176009915)

[**What is the ETCD?** 7](#_Toc176009916)

[Worker Node (formerly minions) 7](#_Toc176009917)

[**Container runtime** 8](#_Toc176009918)

[**Kubelet** 8](#_Toc176009919)

[**Kube-proxy** 8](#_Toc176009920)

[**Pods** 8](#_Toc176009921)

[Difference between Docker Swarm and Kubernetes 9](#_Toc176009922)

[Basic building blocks 9](#_Toc176009923)

[Pods 9](#_Toc176009924)

[Why do we need Pods? 10](#_Toc176009925)

[How containers communicate with each other inside same pod? 10](#_Toc176009926)

[Pods states: 11](#_Toc176009927)

[Controllers 11](#_Toc176009928)

[Services 12](#_Toc176009929)

[Creating Kubernetes Cluster 20](#_Toc176009930)

[Kubernetes Clusters 20](#_Toc176009931)

[Namespace 21](#_Toc176009932)

[Control Plane 21](#_Toc176009933)

[Nodes 21](#_Toc176009934)

[Deploy an App 23](#_Toc176009935)

[Kubernetes Deployments 23](#_Toc176009936)

[Explore an App 25](#_Toc176009937)

[Kubernetes Pods 25](#_Toc176009938)

[Kubernetes Nodes 26](#_Toc176009939)

[Expose an App 27](#_Toc176009940)

[Kubernetes Services 27](#_Toc176009941)

[Annexure I 28](#_Toc176009942)

[[PODs]: Commands & Outputs 28](#_Toc176009943)

[[ReplicaSets]: Commands & Outputs 31](#_Toc176009944)

[[Deployment]: Commands & Outputs 36](#_Toc176009945)

[Annexure II 36](#_Toc176009946)

<https://kubernetes.io/docs/tutorials/kubernetes-basics/_print/>

# What is Containerization ?

- It involves encapsulating or packaging up software code and all its dependencies so that it can run uniformly and consistently on any infrastructure

- Container is an image with an execution environment and having set of instructions

- Containerization allows developers to create and deploy applications faster and more securely

- Containers are often referred to as “lightweight,” meaning they share the machine’s operating system kernel and do not require the overhead of associating an operating system within each application.

- Containers are inherently smaller in capacity than a VM and require less start-up time, allowing far more containers to run on the same compute capacity as a single VM.

- Class --> Image and object --> container

- Docker engine: runtime packaging tool on Docker hosts

- Docker hub --> hub where images are stored

- Benefits: portable and packaged in standard way

- This approach enables the decoupling of application from underlying infrastructure and thus improves business agility, application security and operating environments.

## Which applications are the best candidates for containerization?

Applications that are **stateless** (i.e., pass all information needed to process with the request) and **un-clustered** (i.e., designed to run on a single machine rather than requiring a server cluster to operate) such as network daemons and web server components.

Containers are designed to run only a single process per container (unless the

process itself spawns child processes). If you run multiple unrelated processes in a

single container, it is your responsibility to keep all those processes running, manage

their logs, and so on.

For example, you’d have to include a mechanism for automatically

restarting individual processes if they crash. Also, all those processes would

log to the same standard output, so you’d have a hard time figuring out what process

logged what.

Therefore, you need to run each process in its own container. That’s how Docker

and Kubernetes are meant to be used.

# What is Kubernetes ?

Kubernetes in an open source container management(orchestration) tool hosted by Cloud Native Computing Foundation (CNCF).

It’s container management responsibilities includes:

* container deployment,
* scaling & descaling of containers &
* container load balancing.
* Kubernetes always run processes in containers and each container is much like an isolated machine

Note: *Kubernetes is not a containerization platform. It is a multi-container management solution.*

- Kubernetes is a platform to schedule and run containers on cluster of VMs, bare metal, cloud

- Kubernetes can be used with Docker, where Docker can be used to develop and build applications and then Kubernetes can be used to run these applications on infrastructure

# Features of Kubernetes

- multi-host container scheduling

- done by kube-scheduler

- assigns pods to nodes at runtime

- check resources, quality of services etc before scheduling

- Scalability & Availability

- masters can be deployed in highly available configurations

- multi-region deployments available

- supports 5000 nodes clusters

- 150000 total pods

- pods can be horizontally scaled

- Plug and play architecture, which allows to extend architecture whenever needed

- Registration

- nodes can be registered to Kubernetes master

- Service Discovery

- Services can be automatically detected via DNS

- persistent storage

- much requested and important feature for containers

- Pods can use persistent volume to store data

- Data retained across pod restarts and crashes

- Application Upgrade and Downgrade

- Based on need the applications can be upgraded or downgraded

- Logging

- application monitoring

- health check

- inbuilt logging framework

# Kubernetes Architecture:

Kubernetes implements a cluster computing background, everything works from inside a ***Kubernetes Cluster***. This cluster is hosted by one node acting as the ‘master’ of the cluster, and other nodes as ‘nodes’ which do the actual ‘containerization‘

Diagram

Description automatically generated

Master controls the cluster, and the nodes in it. It ensures the execution only happens in nodes and coordinates the act.

Nodes host the containers; in-fact these Containers are grouped logically to form Pods. Each node can run multiple such Pods, which are a group of containers, that interact with each other, for a deployment.

Replication Controller is Master’s resource to ensure that the requested no. of pods are always running on nodes.

Service is an object on Master that provides load balancing across a replicated group of Pods.

Diagram

Description automatically generated

Kubernetes Architecture has the following main components:

* Master nodes
* Worker/Slave nodes
* Distributed key-value store(etcd.)

## Master node

It is the entry point for all administrative tasks which is responsible for managing the Kubernetes cluster. There can be more than one master node in the cluster to check for fault tolerance. More than one master node puts the system in a High Availability mode, in which one of them will be the main node which we perform all the tasks.

For managing the cluster state, it uses**etcd** in which all the master nodes connect to it.

Diagram

Description automatically generated

### **API server**

* Performs all the administrative tasks through the API server within the master node.
* In this REST commands (runs over HTTP/HTTPs over JSON) are sent to the API server which validates and processes the requests.
* After requesting, the resulting state of the cluster is stored in the distributed key-value store.

### **Scheduler**

* The scheduler schedules the tasks to slave nodes. It stores the resource usage information for each slave node.
* It schedules the work in the form of Pods and Services.
* Before scheduling the task, the scheduler also takes into account the quality of the service requirements, data locality, affinity, anti-affinity, etc.

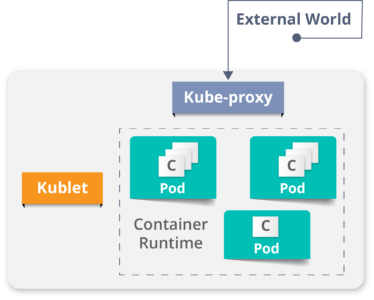
### **Controller manager**

* Also known as **controllers**.
* It is a daemon which regulates the Kubernetes cluster which manages the different non-terminating control loops.
* It also performs lifecycle functions such as namespace creation and lifecycle, event garbage collection, terminated-pod garbage collection, cascading-deletion garbage collection, node garbage collection, etc.
* Basically, a controller watches the desired state of the objects it manages and watches their current state through the API server. If the current state of the objects it manages does not meet the desired state, then the control loop takes corrective steps to make sure that the current state is the same as the desired state.

### **What is the ETCD?**

* etcd is a distributed key-value store which stores the cluster state.
* It can be part of the Kubernetes Master, or, it can be configured externally.
* etcd is written in the Go programming language. In Kubernetes, besides storing the cluster state (based on the **Raft Consensus Algorithm**) it is also used to store configuration details such as subnets, ConfigMaps, Secrets, etc.
* A raft is a consensus algorithm designed as an alternative to Paxos. The Consensus problem involves multiple servers agreeing on values; a common problem that arises in the context of replicated state machines. Raft defines three different roles (Leader, Follower, and Candidate) and achieves consensus via an elected leader

## **Worker Node (formerly minions)**



It is a physical server or you can say a VM which runs the applications using Pods (**a pod scheduling unit**) which is controlled by the master node. On a physical server (worker/slave node), pods are scheduled. For accessing the applications from the external world, we connect to nodes.

### **Container runtime**

* To run and manage a container’s lifecycle, we need a **container runtime**on the worker node.
* Sometimes, Docker is also referred to as a container runtime, but to be precise, Docker is a platform which uses **containers**as a container runtime.

### **Kubelet**

* It is an agent which communicates with the Master node and executes on nodes or the worker nodes. It gets the Pod specifications through the API server and executes the containers associated with the Pod and ensures that the containers described in those Pod are running and healthy.

### **Kube-proxy**

* Kube-proxy runs on each node to deal with individual host sub-netting and ensure that the services are available to external parties.
* It serves as a network proxy and a load balancer for a service on a single worker node and manages the network routing for TCP and UDP packets.
* It is the network proxy which runs on each worker node and listens to the API server for each Service endpoint creation/deletion.
* For each Service endpoint, kube-proxy sets up the routes so that it can reach to it.

### **Pods**

A pod is one or more containers that logically go together. Pods run on nodes. Pods run together as a logical unit. So they have the same shared content. They all share the same IP address but can reach other Pods via localhost, as well as shared storage. Pods don’t need to all run on the same machine as containers can span more than one machine. One node can run multiple pods.

# Difference between Docker Swarm and Kubernetes

Graphical user interface, text, application, chat or text message

Description automatically generated

# Basic building blocks

- Requirements

- Each node must have a kubelet running, which is an agent to for managing the node and communicating with control plane

- Container tooling like docker

- kube-proxy process running

- Supervisord process running that restart components

- in production we should use 3 node clusters at least to run Kubernetes

|  |
| --- |
| ainani@ainani-a01 demo % kubectl get nodes  NAME STATUS ROLES AGE VERSION  minikube Ready control-plane,master 6d2h v1.23.3  ainani@ainani-a01 demo % |

- Minikube

- lightweight Kubernetes implementation that creates a VM on local m/c and deploys a simple cluster containing only one node

## Pods

* Pod is a collection of single or multiple containers in same node
* Containers inside pods can communicate with each other using pod’s IP
* Note that it’s pod not containers who will be assigned with IP
* This pod’s IP is used by containers to communicate each other’s.
* Pod is a simplest unit that can be interacted with
* For simplicity, we can understand that pod will act as VM and containers inside that pod will act as processes. Which are sharing the same network and resources of pod (acting as VM here)

Diagram

Description automatically generated

## Why do we need Pods?

As we are not supposed to group multiple processes into a single container, it’s

obvious you need another higher-level construct that will allow you to bind containers

together and manage them as a single unit. This is the reasoning behind pods.

A pod of containers allows you to run closely related processes together and provide

them with (almost) the same environment as if they were all running in a single

container, while keeping them somewhat isolated. This way, you get the best of both

worlds. You can take advantage of all the features containers provide, while at the

same time giving the processes the illusion of running together.

As we know that containers are completely isolated from

each other, then how they will interact with each other in same pod.

Kubernetes achieves this by configuring Docker to have all containers of a pod share the same set of Linux namespaces instead of each container having its own set.

Because all containers of a pod run under the same Network and UTS namespaces

(we’re talking about Linux namespaces here), they all share the same hostname and

network interfaces. Similarly, all containers of a pod run under the same IPC namespace

and can communicate through IPC.

## How containers communicate with each other inside same pod?

Containers in a pod run in the same Network namespace, they share the same IP address and port space. This means processes running in containers of the same pod need to take care not to bind to the same port numbers or they’ll run into port conflicts. But this only concerns containers in the same pod. Containers of different pods can never run into port conflicts, because each pod has a separate port space.   
All the containers in a pod also have the same loopback network interface, so a container can communicate with other containers in the same pod through localhost.

|  |
| --- |
| ainani@ainani-a01 demo % kubectl run sise --image=quay.io/openshiftlabs/simpleservice:0.5.0 --port=9876  pod/sise created  ainani@ainani-a01 demo %  ainani@ainani-a01 demo % kubectl get pods  NAME READY STATUS RESTARTS AGE  sise 1/1 Running 0 4m3s  ainani@ainani-a01 demo %  # image sise correspond to curl command  ainani@ainani-a01 demo % kubectl exec sise -t -- curl -s localhost:9876/info  {"host": "localhost:9876", "version": "0.5.0", "from": "127.0.0.1"}**%** ainani@ainani-a01 demo % |

- Requirements:

- docker application container

- Storage resources

- Unique n/w ip

- options that governs how the container(s) should run- Pods are ephemeral and disposable

- Never self-heal and not restarted by scheduler it self

- Never create pods by just themselves

- use higher-level constructs

- Pods are started by some controllers

### Pods states:

- **Pending** - pod had been accepted by Kubernetes system but a container had not been created yet

- **Running** - pod had been scheduled over node, all it's containers are created and at least one of it's container is in running state

- **Succeeded** - all the containers in pods are exited with exist status 0

- **Failed** - all the containers in pod are executed and at least one container is failed with non zero exit status

- **CrashLoopBackOff** - Container is failed to start due to some region and Kubernetes tried over and over to start the pod

- Kubernetes Probes the Pods to check health of Pods – Means checking that application inside pod is running or not.

- Pods are Ephemeral – Means Pod can’t be redeployed. Once a pod died, it can’t be redeployed. Kubernetes had to create new pod

## Controllers

- Controllers creates Pods for system

- Controllers keep systems in desired state (mainly Pods)

- application reliability - multiple instances of applications running prevent problems if one or more instance of application fails

- Scaling - if pod experience larger volume data Kubernetes allows the pod to scale

- Load balancing - when multiple pods are running, traffic can be distributed to other

pods doesn't overload one single pod

- Kinds of controllers

- ReplicaSets

- Deployments

- DaemonSets

- Jobs

- Services

- ReplicaSet

- Ensure that a specified no. of replicas for a pod are running at all times

- if a pod crashed then replicaSet will ensure to start a new pod to ensure the no. of replica pod should be same as declared

- ReplicaSet cannot be declared by itself and should be used in deployment set

- Deployments

- Deployment controller provides declarative updates for pods and ReplicaSets

- This mean you can defined a desired state of deployment in a YAML file and deployment controller will align the actual state to match

- Deployment sets can be defined to create new replicaSet or replace the existing once

- Deployment manages ReplicaSet, which manages pods

- Pod management - running a replica-set allows us to deploy a number of pods and check their status as a single unit

- Scaling of ReplicaSet : scales out the pods and allows for the deployment to handle more traffic

- Pause a deployment make changes and resume the deployment set

- A new replicaSet will be started every-time after pause-resume

- Pausing a deployment means only updates are paused but traffic is still running

- Status : easy way to check health of pods

- DaemonSets:

- Ensures that all nodes run a copy of a specific pod

- As nodes are added or removed from the cluster, a DaemonSet will add or remove the required pods

- Deleting a DaemonSet will also cleanup the all the pods it's created

- a typical use case of DaemonSet is to run a single log aggregator or a monitoring agent on a node

- Jobs:

- Supervisor process for pods carrying out batch jobs

- Run individual processes that run once and complete successfully

- Jobs are run in cronjob to run a specific process at specific time

## Services

- Communication b/w one set of deployments with another

- When a service is created it's assigned with unique IP address which will never change throughout out the lifetime of service

- Pods are then configured to talk to services and rely on service IP for any communication

- Services are really important because they allow one set of pods to communicate with other set of pods

- in simple form, it's best practice to use a service when you allow one deployment to talk to another deployment

- If frontend pods needs to talk to backend pods then it should communicate via Service IP, because backend pods IP may changes anytime

- Internal Service : IP is only reachable within cluster, also known as cluster IP in K8s

- External Service : Endpoint available through node ip:port (called NodePort)

- LoadBalancer : Exposes application to internet with a load balancer

### Labels, selectors and namespaces

Labels:

- Labels are key/value pairs that are attached to objects like pods, services and deployments.

- Labels are for users of Kubernetes to identify attributes for objects

- Label keys are unique for objects

- Labels are generally used with selectors

Selectors:

- Equality selectors : Equals and not equals

- Set based selectors : In, NOTIN and EXISTS

- labels and selectors are used with kubectl to lister or filter objects

Namespaces:

- Great for large enterprises

### Kubelet and KubeProxy:

# Kubelet

- Kubelet is "Kubernetes node agent" that runs on each node

- communicate with API server to see if pods have been assigned to nodes

- Executes pods containers via a container engine

- mounts and runs pod volumes and secrets

- Execute health check to identify pod/node status

- Podspec: YAML file that describe a pod

- Kubelet takes set of Prodspecs and ensures that the containers in those Prodspec are running and healthy

- Kubelet only manages the containers createdby API server - not any other one

# KubeProxy : N/w Proxy

- process run on all worker nodes

- Reflects services as defined on each node and can to simple n/w stream or round robin forwarding across a set of backends

3 modes:

- User space mode - most common

- IPtables mode

- IPVs mode

- Services are defnied against the API server, kube-proxy watches the API server for the addition and removal of services

- For each new service, kube-proxy opens a randomly choosen port on the local node

- connections mage to the choosen port are proxied to one of the corresponding back-end pods

### Install

- Docker and Hypervisor should be installed already for Kubernetes

- Install Docker desktop for Mac/windows

- Install hypervisor

### Hello World

- minikube start

😄 minikube v1.25.2 on Darwin 12.2.1 (arm64)

✨ Using the docker driver based on existing profile

👍 Starting control plane node minikube in cluster minikube

🚜 Pulling base image ...

🏃 Updating the running docker "minikube" container ...

🐳 Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...

▪ kubelet.housekeeping-interval=5m

🔎 Verifying Kubernetes components...

▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5

🌟 Enabled addons: storage-provisioner, default-storageclass

🏄 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default

- kubectl get nodes

NAME STATUS ROLES AGE VERSION

minikube Ready control-plane,master 13m v1.23.3

- kubectl create -f /Users/ainani/Downloads/Ex\_Files\_Learning\_Kubernetes\_Upd/Exercis\_Files/03\_04/helloworld.yaml

deployment.apps/helloworld created

- kubectl get deployments

NAME READY UP-TO-DATE AVAILABLE AGE

helloworld 1/1 1 1 22m

- kubectl get rs

NAME DESIRED CURRENT READY AGE

helloworld-d7c6dd56 1 1 1 22m

- kubectl get pods

NAME READY STATUS RESTARTS AGE

helloworld-d7c6dd56-4s5ds 1/1 Running 0 22m

- kubectl get all

NAME READY STATUS RESTARTS AGE

pod/helloworld-d7c6dd56-4s5ds 1/1 Running 0 31m

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 63m

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/helloworld 1/1 1 1 31m

NAME DESIRED CURRENT READY AGE

replicaset.apps/helloworld-d7c6dd56 1 1 1 31m

- kubectl expose deployment helloworld --type=NodePort

service/helloworld exposed

- kubectl get all

NAME READY STATUS RESTARTS AGE

pod/helloworld-d7c6dd56-4s5ds 1/1 Running 0 44m

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/helloworld NodePort 10.100.70.103 <none> 80:32673/TCP 28s

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 77m

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/helloworld 1/1 1 1 44m

NAME DESIRED CURRENT READY AGE

replicaset.apps/helloworld-d7c6dd56 1 1 1 44m

- kubectl get services

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 57m

- minikube service <service name> --> see application up and running

-

### Breaking hello World

-kubectl get deployment

NAME READY UP-TO-DATE AVAILABLE AGE

helloworld 1/1 1 1 135m

- kubectl get deployment/helloworld -o yaml

apiVersion: apps/v1

kind: Deployment

metadata:

annotations:

deployment.kubernetes.io/revision: "1"

creationTimestamp: "2022-06-16T06:02:17Z"

generation: 1

name: helloworld

namespace: default

resourceVersion: "1862"

uid: 02040503-08f0-4a68-ab84-dde4179b93cb

spec:

progressDeadlineSeconds: 600

replicas: 1

revisionHistoryLimit: 10

selector:

matchLabels:

app: helloworld

strategy:

rollingUpdate:

maxSurge: 25%

maxUnavailable: 25%

type: RollingUpdate

template:

metadata:

creationTimestamp: null

labels:

app: helloworld

spec:

containers:

- image: karthequian/helloworld:latest

imagePullPolicy: Always

name: helloworld

ports:

- containerPort: 80

protocol: TCP

resources: {}

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: File

dnsPolicy: ClusterFirst

restartPolicy: Always

schedulerName: default-scheduler

securityContext: {}

terminationGracePeriodSeconds: 30

status:

availableReplicas: 1

conditions:

- lastTransitionTime: "2022-06-16T06:02:32Z"

lastUpdateTime: "2022-06-16T06:02:32Z"

message: Deployment has minimum availability.

reason: MinimumReplicasAvailable

status: "True"

type: Available

- lastTransitionTime: "2022-06-16T06:02:17Z"

lastUpdateTime: "2022-06-16T06:02:32Z"

message: ReplicaSet "helloworld-d7c6dd56" has successfully progressed.

reason: NewReplicaSetAvailable

status: "True"

type: Progressing

observedGeneration: 1

readyReplicas: 1

replicas: 1

updatedReplicas: 1

- kubectl get service/helloworld -o yaml

apiVersion: v1

kind: Service

metadata:

creationTimestamp: "2022-06-16T06:46:47Z"

name: helloworld

namespace: default

resourceVersion: "3516"

uid: d6f61616-a9c4-4262-a9d1-3b17d9094d0b

spec:

clusterIP: 10.100.70.103

clusterIPs:

- 10.100.70.103

externalTrafficPolicy: Cluster

internalTrafficPolicy: Cluster

ipFamilies:

- IPv4

ipFamilyPolicy: SingleStack

ports:

- nodePort: 32673

port: 80

protocol: TCP

targetPort: 80

selector:

app: helloworld

sessionAffinity: None

type: NodePort

status:

loadBalancer: {}

### Scaling Hello world

- kubectl scale --replicas=3 deploy/<name>

- kubectl get deploy/<name>

### Add, Change and Delete labels

- kubectl get pods --show-labels

- kubectl label pod/<name> app=<new label name> --overwrite

- kubectl label pod/<name> app- => remove label

### Working with labels

- kubectl get pods --selector env=production

- kubectl get pods --selector dev-lead=name,env=staging

- kubectl get pods --selector dev-lead!=name,env=staging --> NOT EQUAL

- kubectl get pods -l release-version in (1.0.2.0)'

- kubectl get pods -l release-version notin (1.0.2.0)'

- kubectl delete pods -l dev-lead=<name>

- kubectl get pods --show-labels

### Application health checks

- kubectl describe po/<pod name>

### Upgrades

- kubectl create -f <name> --record

--> record roll out history

- kubectl set image deploymnet/<name> helloworld=<new name of image>

- kubectl rollout undo deployment/<name>

### Troubleshooting

- kubectl describe deployment <deploy name>

- kebectl describe pod <pod name>

- kubectl logs <pod name>

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

- kubectl exec -it <pod name> -c <pod2 name>/bin/bash --> in case multiple pods

### Dashboard

- Allow to visualize and monitor cluster for operational purpose

- minikube addons list

- minikube dashboard

- minikube addons enable heapster

### Configurations

- kubectl create configmap logger --from-literal=log\_level=debug

- kubectl get configmaps

- kubectl get configmap/logger -o yaml

- kubectl logs <pod name>

### Secret data

- kubectl create secret generic apikey --from-literal=api\_key=1234567

- kubectl get secret apikey -o yaml

### Running jobs

- kubectl get jobs

- kubectl get cronjobs

### Running stateful applications

- kubectl get daemonset

- stateful sets manages the deployment and scaling for set of pods

- provides guarantees and ordering of pods

- kubectl get statefulsets

### Kubernetes in Production

- using kubeadm tool

### Monitoring and Logging

- kubectl logs

- Stdout

- cAdvisor

- Heapster

- Prometheus

### Authentication & Authorization

- normal users - intereact with systems

- service accounts - account managed by K8s API

- username, UID, group, extra fields

- Authenticate checks that a user have access to system or not

- Authorization checks that user can perform an action in the system or not

Methods:

- client cert

- Token file

- OpenID connect

- Webhook Tokens

### kind

- brew install kind

- kind version

- kind create cluster --name test1

Creating cluster "test1" ...

✓ Ensuring node image (kindest/node:v1.24.0) 🖼

✓ Preparing nodes 📦

✓ Writing configuration 📜

✓ Starting control-plane 🕹️

✓ Installing CNI 🔌

✓ Installing StorageClass 💾

Set kubectl context to "kind-test1"

You can now use your cluster with:

kubectl cluster-info --context kind-test1

Have a nice day! 👋

- kubectl get namespaces

NAME STATUS AGE

default Active 4m31s

kube-node-lease Active 4m33s

kube-public Active 4m33s

kube-system Active 4m33s

local-path-storage Active 4m27s

- kubectl get pods --namespace kube-system

NAME READY STATUS RESTARTS AGE

coredns-6d4b75cb6d-m2dfw 1/1 Running 0 4m35s

coredns-6d4b75cb6d-t4zc9 1/1 Running 0 4m35s

etcd-test1-control-plane 1/1 Running 0 4m47s

kindnet-p496k 1/1 Running 0 4m35s

kube-apiserver-test1-control-plane 1/1 Running 0 4m47s

kube-controller-manager-test1-control-plane 1/1 Running 0 4m47s

kube-proxy-x6g8r 1/1 Running 0 4m35s

kube-scheduler-test1-control-plane 1/1 Running 0 4m47s

# Creating Kubernetes Cluster

## Kubernetes Clusters

**A Kubernetes cluster is a set of nodes that run containerized applications. Containerizing applications packages an app with its dependences and some necessary services. They are more lightweight and flexible than virtual machines.**

* Kubernetes clusters allow containers to run across multiple machines and environments: virtual, physical, cloud-based, and on-premises.
* Kubernetes containers are not restricted to a specific operating system, unlike virtual machines. Instead, they are able to share operating systems and run anywhere
* Kubernetes clusters are comprised of one master node and a number of worker nodes. These nodes can either be physical computers or virtual machines, depending on the cluster
* There must be a minimum of one master node and one worker node for a Kubernetes cluster to be operational.
* For production and staging, the cluster is distributed across multiple worker nodes.
* For testing, the components can all run on the same physical or virtual node.
* Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way.
* Kubernetes is an open-source platform and is production-ready.

### Namespace

A **namespace** is a way for a Kubernetes user to organize many different clusters within just one physical cluster. Namespaces enable users to divide cluster resources within the physical cluster among different teams via resource quotas. For this reason, they are ideal in situations involving complex projects or multiple teams.

A Kubernetes cluster consists of two types of resources:

* The Control Plane coordinates the cluster
* Nodes are the workers that run applications

Diagram

Description automatically generated

### Control Plane

- Responsible to manage the cluster and nodes that are used to host the running apps

- co-ordinates all activities in a cluster like scheduling apps, scaling apps and rolling out new updates etc

### Nodes

* **A node is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster.**
* Each node has a Kubelet, which is an agent for managing the node and communicating with the Kubernetes control plane.
* The node should also have tools for handling container operations, such as containerd or Docker.
* A Kubernetes cluster that handles production traffic should have a minimum of three nodes because if one node goes down, both an etcd member and a control plane instance are lost, and redundancy is compromised. You can mitigate this risk by adding more control plane nodes.
* **The nodes communicate with the control plane using the**[**Kubernetes API**](https://kubernetes.io/docs/concepts/overview/kubernetes-api/), which the control plane exposes.
* End users can also use the Kubernetes API directly to interact with the cluster.

When you deploy applications on Kubernetes, you tell the control plane to start the application containers.

The control plane schedules the containers to run on the cluster's nodes.

* A Kubernetes cluster can be deployed on either physical or virtual machines. To get started with Kubernetes development, you can use Minikube. Minikube is a lightweight Kubernetes implementation that creates a VM on your local machine and deploys a simple cluster containing only one node. Minikube is available for Linux, macOS, and Windows systems. The Minikube CLI provides basic bootstrapping operations for working with your cluster, including start, stop, status, and delete.

|  |
| --- |
| ainani@ainani-a01 demo % minikube version  minikube version: v1.25.2  commit: 362d5fdc0a3dbee389b3d3f1034e8023e72bd3a7  ainani@ainani-a01 demo % minikube start  😄 minikube v1.25.2 on Darwin 12.2.1 (arm64)  🎉 minikube 1.26.0 is available! Download it: https://github.com/kubernetes/minikube/releases/tag/v1.26.0  💡 To disable this notice, run: 'minikube config set WantUpdateNotification false'  ✨ Using the docker driver based on existing profile  👍 Starting control plane node minikube in cluster minikube  🚜 Pulling base image ...  🏃 Updating the running docker "minikube" container ...  🐳 Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...  ▪ kubelet.housekeeping-interval=5m  🔎 Verifying Kubernetes components...  ▪ Using image quay.io/rhdevelopers/core-dns-patcher  ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5  ▪ Using image alpine:3.11  ▪ Using image gcr.io/google\_containers/pause:3.1  ╭──────────────────────────────────────────────────────────────────────────────────────────────────────╮  │ │  │ Registry addon with docker driver uses port 52417 please use that instead of default port 5000 │  │ │  ╰──────────────────────────────────────────────────────────────────────────────────────────────────────╯  📘 For more information see: https://minikube.sigs.k8s.io/docs/drivers/docker  ▪ Using image registry:2.7.1  ▪ Using image gcr.io/google\_containers/kube-registry-proxy:0.4  🔎 Verifying registry addon...  🌟 Enabled addons: storage-provisioner, registry-aliases, default-storageclass, registry  🏄 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default  ainani@ainani-a01 demo % kubectl version  WARNING: This version information is deprecated and will be replaced with the output from kubectl version --short. Use --output=yaml|json to get the full version.  Client Version: version.Info{Major:"1", Minor:"24", GitVersion:"v1.24.2", GitCommit:"f66044f4361b9f1f96f0053dd46cb7dce5e990a8", GitTreeState:"clean", BuildDate:"2022-06-15T14:14:10Z", GoVersion:"go1.18.3", Compiler:"gc", Platform:"darwin/arm64"}  Kustomize Version: v4.5.4  Server Version: version.Info{Major:"1", Minor:"23", GitVersion:"v1.23.3", GitCommit:"816c97ab8cff8a1c72eccca1026f7820e93e0d25", GitTreeState:"clean", BuildDate:"2022-01-25T21:19:12Z", GoVersion:"go1.17.6", Compiler:"gc", Platform:"linux/arm64"}  ainani@ainani-a01 demo % kubectl cluster-info  Kubernetes control plane is running at https://127.0.0.1:52418  CoreDNS is running at https://127.0.0.1:52418/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy  To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.  ainani@ainani-a01 demo % kubectl get nodes  NAME STATUS ROLES AGE VERSION  minikube Ready control-plane,master 6d22h v1.23.3  ainani@ainani-a01 demo % |

# Deploy an App

## Kubernetes Deployments

* Containerized apps are deployed over Kubernetes cluster
* To do so, user creates Deployment configuration
* The Deployment instructs Kubernetes how to create and update instances of your application.
* Once you've created a Deployment, the Kubernetes control plane schedules the application instances included in that Deployment to run on individual Nodes in the cluster.
* Once the application instances are created, a Kubernetes Deployment Controller continuously monitors those instances.
* If the Node hosting an instance goes down or is deleted, the Deployment controller replaces the instance with an instance on another Node in the cluster.
* **This provides a self-healing mechanism to address machine failure or maintenance.**

Diagram

Description automatically generated

You can create and manage a Deployment by using the Kubernetes command line interface, **Kubectl**.

Kubectl uses the Kubernetes API to interact with the cluster.

When you create a Deployment, you'll need to specify the container image for your application and the number of replicas that you want to run. You can change that information later by updating your Deployment

|  |
| --- |
| ainani@ainani-a01 demo % kubectl version  WARNING: This version information is deprecated and will be replaced with the output from kubectl version --short. Use --output=yaml|json to get the full version.  Client Version: version.Info{Major:"1", Minor:"24", GitVersion:"v1.24.2", GitCommit:"f66044f4361b9f1f96f0053dd46cb7dce5e990a8", GitTreeState:"clean", BuildDate:"2022-06-15T14:14:10Z", GoVersion:"go1.18.3", Compiler:"gc", Platform:"darwin/arm64"}  Kustomize Version: v4.5.4  Server Version: version.Info{Major:"1", Minor:"23", GitVersion:"v1.23.3", GitCommit:"816c97ab8cff8a1c72eccca1026f7820e93e0d25", GitTreeState:"clean", BuildDate:"2022-01-25T21:19:12Z", GoVersion:"go1.17.6", Compiler:"gc", Platform:"linux/arm64"}  ainani@ainani-a01 demo % kubectl get nodes  NAME STATUS ROLES AGE VERSION  minikube Ready control-plane,master 6d23h v1.23.3  ainani@ainani-a01 demo % kubectl create deployment kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1  deployment.apps/kubernetes-bootcamp created  ainani@ainani-a01 demo % kubectl get deployments  NAME READY UP-TO-DATE AVAILABLE AGE  kubernetes-bootcamp 1/1 1 1 50s  ainani@ainani-a01 demo % |

# Explore an App

## Kubernetes Pods

* Once user created a Deployment, Kubernetes created a **Pod** to host your application instance.
* A Pod is a Kubernetes abstraction that represents a group of one or more application containers (such as Docker), and some shared resources for those containers. Those resources include:
  + Shared storage, as Volumes
  + Networking, as a unique cluster IP address
  + Information about how to run each container, such as the container image version or specific ports to use
* A Pod models an application-specific "logical host" and can contain different application containers which are relatively tightly coupled.
* For example, a Pod might include both the container with your Node.js app as well as a different container that feeds the data to be published by the Node.js webserver.
* The containers in a Pod share an IP Address and port space, are always co-located and co-scheduled, and run in a shared context on the same Node.
* Pods are the atomic unit on the Kubernetes platform. When we create a Deployment on Kubernetes, that Deployment creates Pods with containers inside them (as opposed to creating containers directly).
* Each Pod is tied to the Node where it is scheduled, and remains there until termination (according to restart policy) or deletion. In case of a Node failure, identical Pods are scheduled on other available Nodes in the cluster.
* *A Pod is a group of one or more application containers (such as Docker) and includes shared storage (volumes), IP address and information about how to run them*

Diagram

Description automatically generated

## Kubernetes Nodes

A Pod always runs on a **Node**. A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster. Each Node is managed by the control plane. A Node can have multiple pods, and the Kubernetes control plane automatically handles scheduling the pods across the Nodes in the cluster. The control plane's automatic scheduling takes into account the available resources on each Node.

Every Kubernetes Node runs at least:

* Kubelet, a process responsible for communication between the Kubernetes control plane and the Node; it manages the Pods and the containers running on a machine.
* A container runtime (like Docker) responsible for pulling the container image from a registry, unpacking the container, and running the application.

Diagram, schematic

Description automatically generated

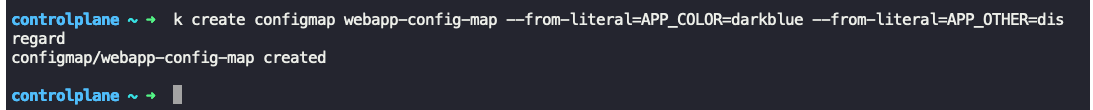
# Expose an App

## Kubernetes Services

* When a worker node dies, the Pods running on the Node are also lost.
* A [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) might then dynamically drive the cluster back to desired state via creation of new Pods to keep your application running.
* A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them.
* Services enable a loose coupling between dependent Pods. A Service is defined using YAML or JSON, like all Kubernetes objects.
* Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service.
* Services allow your applications to receive traffic.

# ConfigMaps

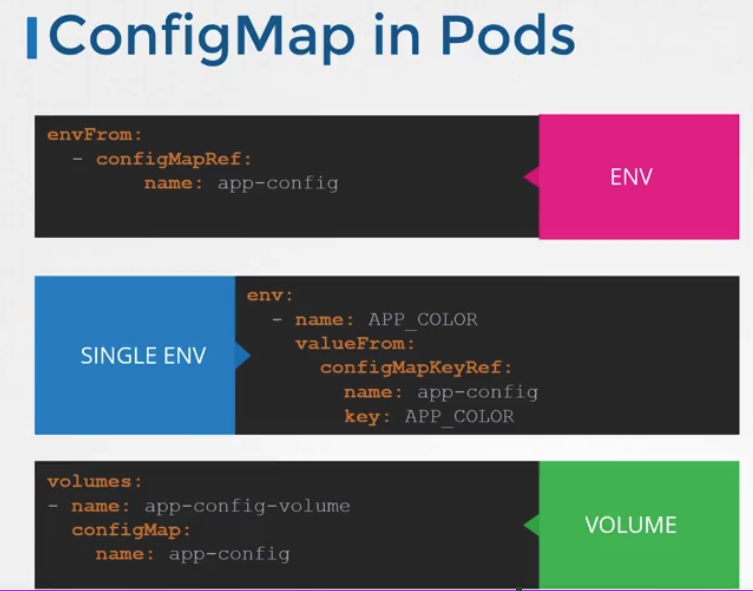
ConfigMaps are used to passed environments variable in pod-definitions’ yaml file.

1. Create ConfigMap
   1. Imperative Way: using Command line directly  
      Syntax:   
       kubectl create configmap <config-name> --from-literal=<key>=<value> --from-literal=<key>=<value>  
        
      Example:  
       kubectl create configmap app-config --from-literal=APP\_COLOR=darkblue --from-literal=APP\_OTHER=disregard  
        
        
      Note: This method become difficult when there are lot of --from-literal, so we generally use another method using file  
        
      Syntax:  
       kubectl create configmap <config-name> --from-file=<path-to-file>  
      Example:  
       kubectl create configmap app-config --from-file=app\_config.properties
   2. Declarative Way: using an config-map.yaml file

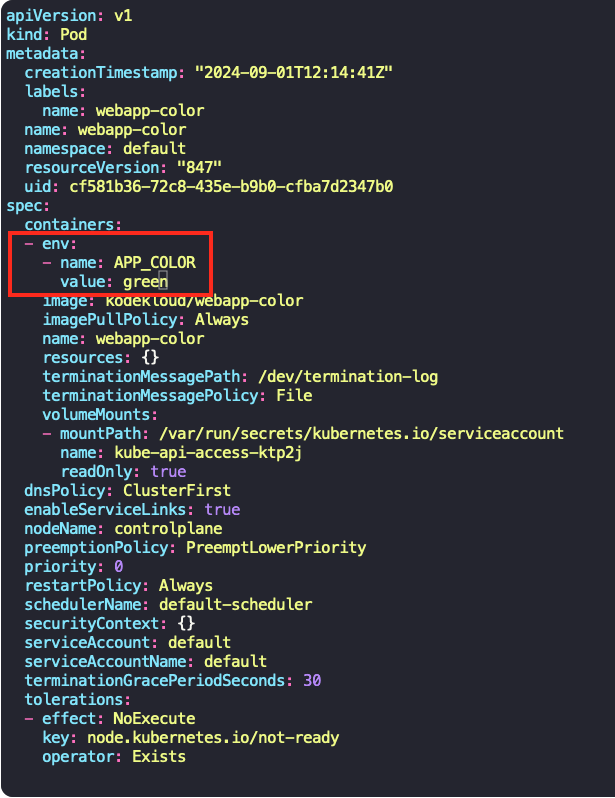
|  |
| --- |
| apiVersion: v1 kind: ConfigMap  metadata:  name: app-config  data:  APP\_COLOR: blue  APP\_MODE: prod |

Syntax: kubectl create -f config-map.yaml

## Injecting ConfigMaps in pod-definition.yaml



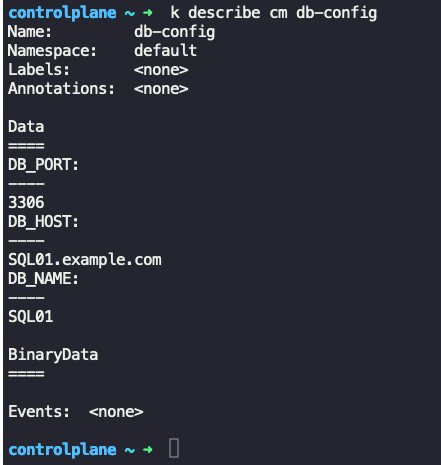
## Practice & Outputs

* Command to find the environments set in pod-definition.yaml  
  Syntax: k describe pod webapp-color  
  Example: k describe pod webapp-color 
* Command to update the env variable
  + Take output of pod definition in .yaml
  + Edit the existing file  
    
  + Delete the existing pod
  + Create new pod using edited file
* Command to get configMaps

Syntax:  
 k get configmaps

k get cm  
Example:  
 

* Command to describe configMap  
  Syntax:   
   k get cm <configmap\_name>

Example:  
 

* Command to create new config map using yaml file

|  |
| --- |
| apiVersion: v1  kind: ConfigMap  metadata:  name: webapp-config-map  data:  APP\_COLOR: darkblue  APP\_OTHER: disregard |

Syntax:  
 k create -f new-cm.yaml  
Example:  
 

* Command to delete configmap  
  Syntax:  
   k delete cm <configmap\_name>  
  Example:  
   
* To use the newly created configMap in pod file  
    
  A screen shot of a computer code

  Description automatically generated

# Annexure I

Create an alias of kubectl command first:

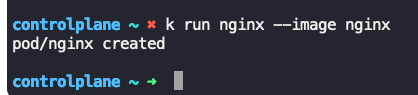
* This will save your time to write longer commands as **kubectl** is general cli command used everywhere

|  |
| --- |
| alias k=kubectl |

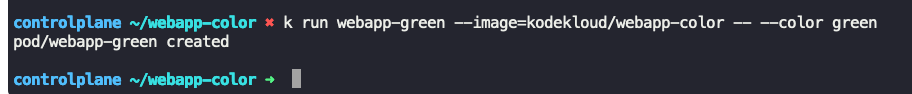
## [PODs]: Commands & Outputs

* Command to create a pod directly without using yaml file:

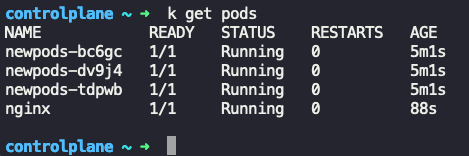
Syntax:   
 k run <pod\_name> --image=<image\_name>  
Example: k run nginx --image nginx



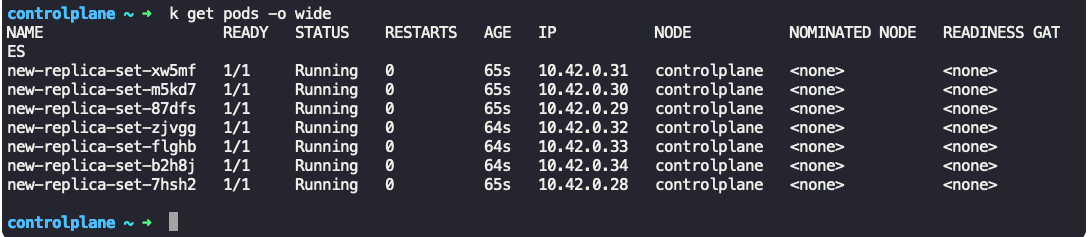
* Command to create a pod directly without using yaml file and passing some additional arguemnts  
  Syntax:

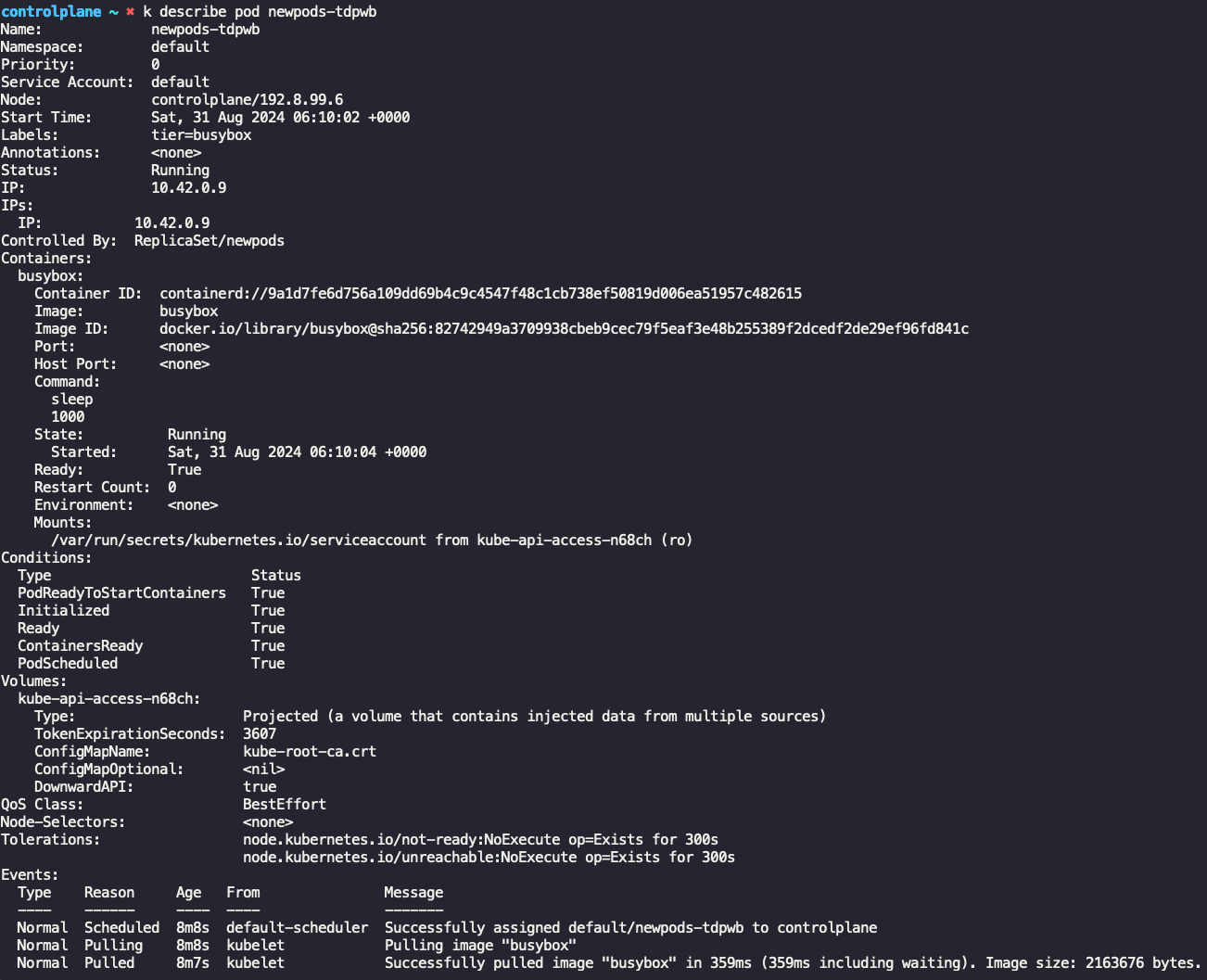
k run <pod\_name> --image=<image\_name> -- <cmd\_args1> <cmd\_args2>  
Example: k run webapp-green --image=kodekloud/webapp-color -- --color green  


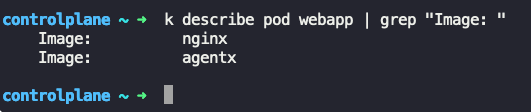
* Command to list(get) all pods  
  Syntax:   
   k get pods  
   k get po  
  Example:

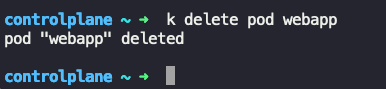


* Command to get detailed output of pods  
  Syntax:  
   k get pods -o wide  
  Example:



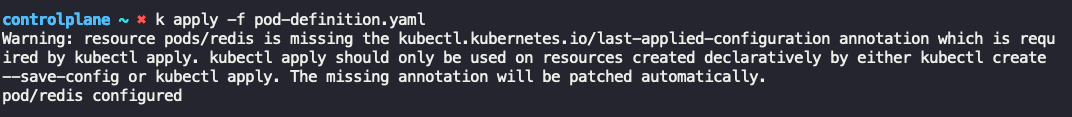
* Command to describe a pod  
  Syntax:  
   k describe pod <pod\_name>  
  Example:  
  

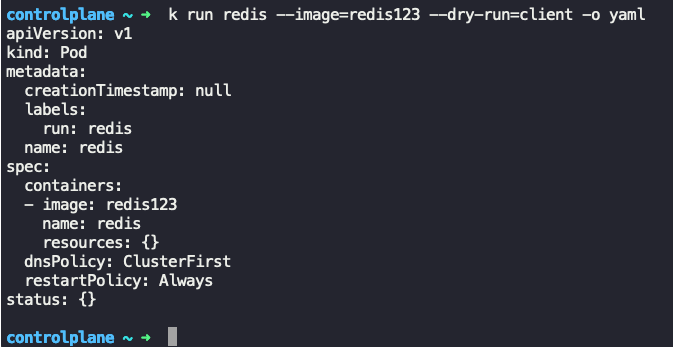
Note: To extract a specific fields use “grep”  
Example: To get image used in pod:  
 k describe pod <pod\_name> | grep “Image: ”  


* Command to delete pod  
  Syntax:  
   k delete pod <pod\_name>  
  Example:  
  
* An example YAML file to create a simple pod

|  |
| --- |
| apiVersion: v1  kind: Pod This is case sensitive, Pod and pod  metadata:  name: redis This name would be used as pod name while creating pod  spec:  containers:  - name: redis  image: redis  command: ["sleep", "5000"] |

|  |
| --- |
| command:   * “sleep” * “1200” Everything is in string in .yaml file even an integer also |

* Command to create a pod (K8 object)  
  Syntax:   
   k create -f <yaml\_file>  
  Example:  
  
* Command to update the existing config in yaml and apply it  
  Syntax:   
   k apply -f <yaml\_file>  
  Example:  
  

Extract the definition of pod to a file using the below command:

Extract the definition of a pod to a file using the below command:

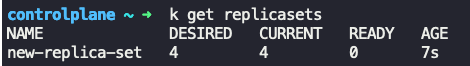
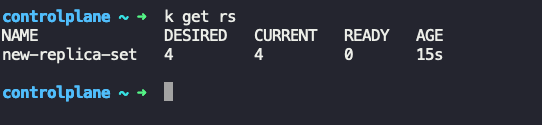
kubectl get pod <pod-name> -o yaml > pod-definition.yaml

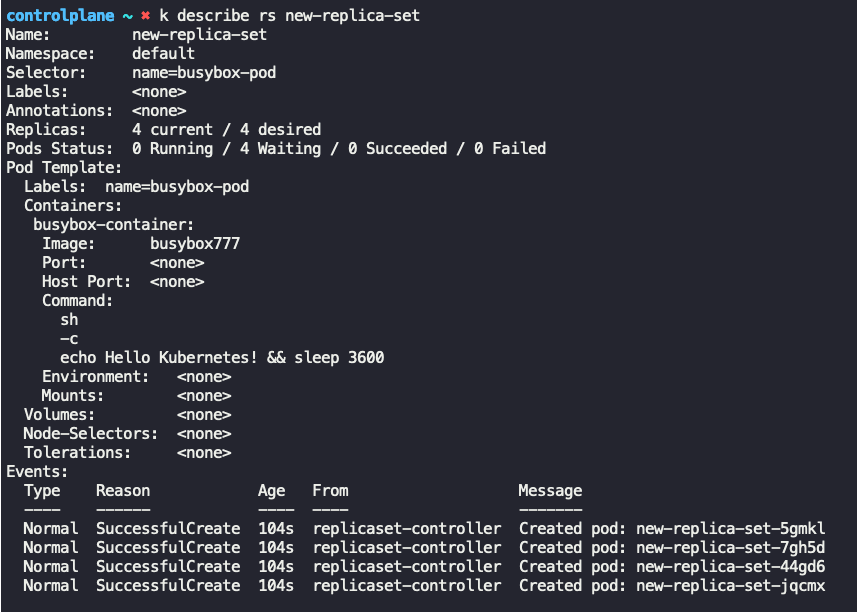
To modify the properties of the pod:

kubectl edit pod <pod-name>

|  |
| --- |
| Please note that only the properties listed below are editable.   * spec.containers[\*].image * spec.initContainers[\*].image * spec.activeDeadlineSeconds * spec.tolerations * spec.terminationGracePeriodSeconds |

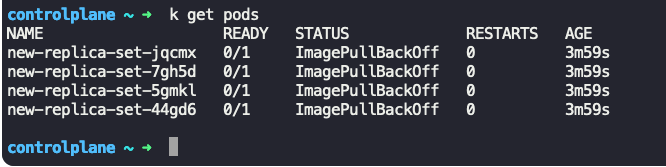
## [ReplicaSets]: Commands & Outputs





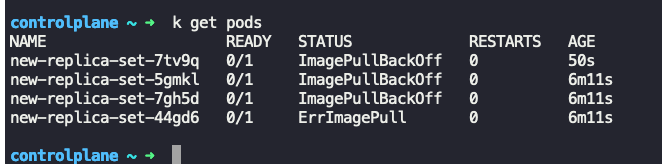
Get pods in replicaset



Delete a pod in



But no. of pods again become 4, because replicaset is configured to have atleast 4 pods



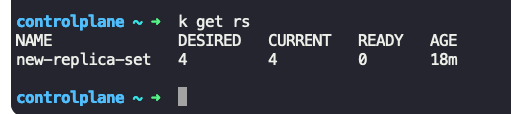
cat /root/replicaset-definition-1.yaml

|  |
| --- |
| apiVersion: apps/v1  kind: ReplicaSet  metadata:  name: replicaset-1  labels:  name: frontend  spec:  replicas: 2  selector:  matchLabels:  tier: frontend  template:  metadata:  labels:  tier: frontend  spec:  containers:  - name: nginx  image: nginx |









To edit the replicaSet,   
k edit rs new-replica-set

|  |
| --- |
| Open yaml in editor mode  Update the yaml  Save it |

Note: Updating yaml will not update the pods image here, user need to either delete the pods or replicasets to make the changes take affect.

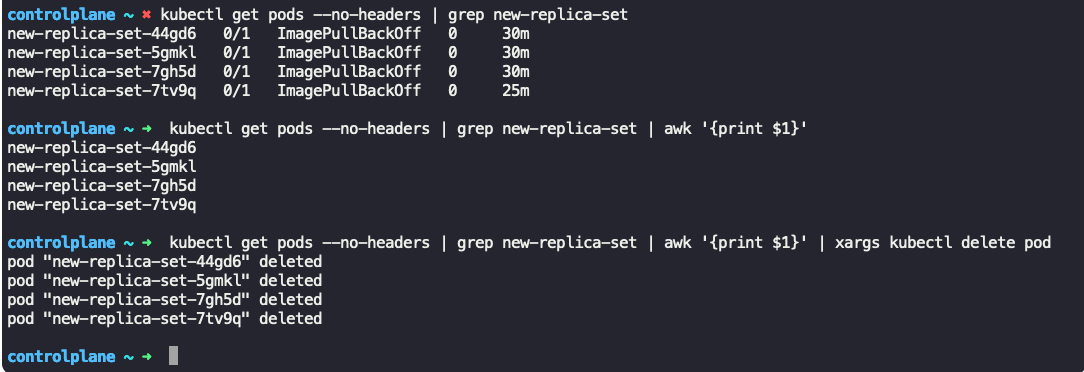
k get rs -o yaml

|  |
| --- |
| apiVersion: v1  items:  - apiVersion: apps/v1  kind: ReplicaSet  metadata:  creationTimestamp: "2024-08-31T07:42:16Z"  generation: 1  name: new-replica-set  namespace: default  resourceVersion: "1140"  uid: 9aae427c-7146-4aa9-984a-618d8c53899b  spec:  replicas: 4  selector:  matchLabels:  name: busybox-pod  template:  metadata:  creationTimestamp: null  labels:  name: busybox-pod  spec:  containers:  - command:  - sh  - -c  - echo Hello Kubernetes! && sleep 3600  image: busybox777  imagePullPolicy: Always  name: busybox-container  resources: {}  terminationMessagePath: /dev/termination-log  terminationMessagePolicy: File  dnsPolicy: ClusterFirst  restartPolicy: Always  schedulerName: default-scheduler  securityContext: {}  terminationGracePeriodSeconds: 30  status:  fullyLabeledReplicas: 4  observedGeneration: 1  replicas: 4  kind: List  metadata:  resourceVersion: "" |

Do note that running create command on same object will give error

|  |
| --- |
|  |

Delete all pods matching same kind of name as wild card:

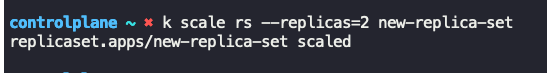


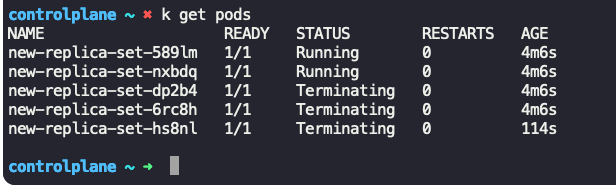
Another method:



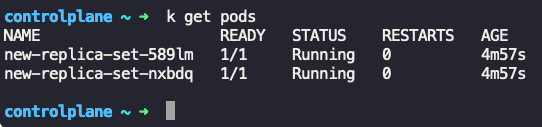


Scale Down the pods in replicaset:



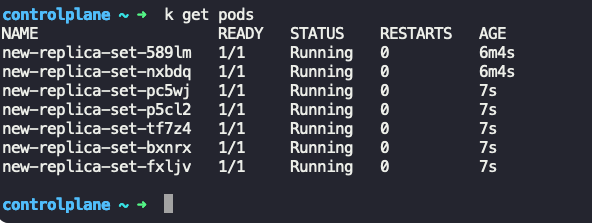
Pods in terminating state:  


After sometime:



Scale Up the pods in replicaset

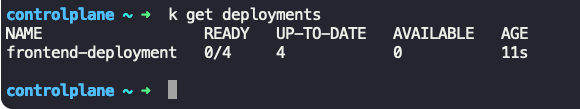




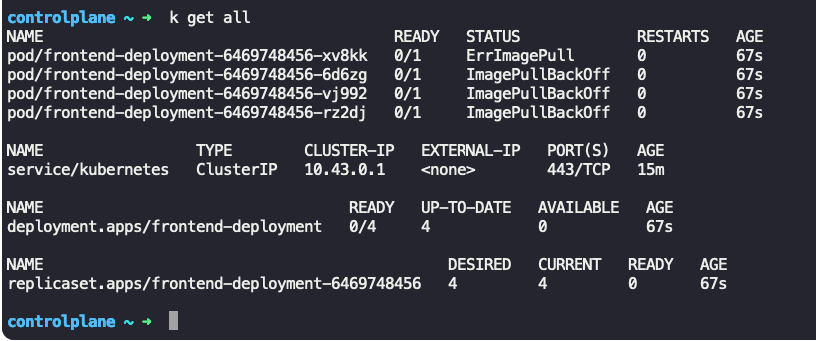
Note: Do note that scale command will update the definition of yaml file also

|  |
| --- |
|  |

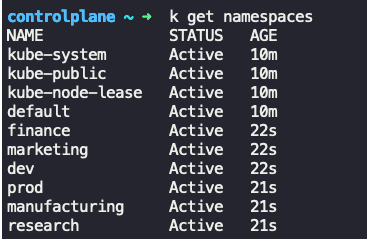
## [Deployment]: Commands & Outputs

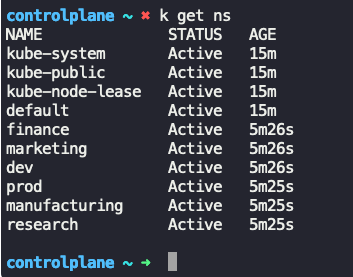


Master command:

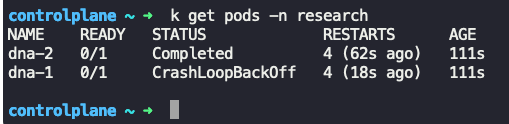


## [Namespaces]: Commands & Outputs

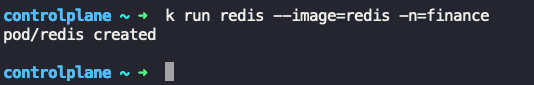


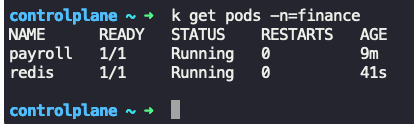


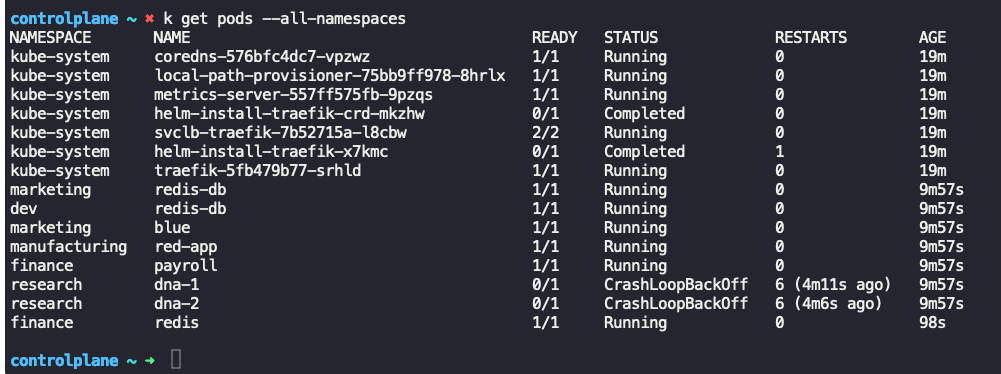


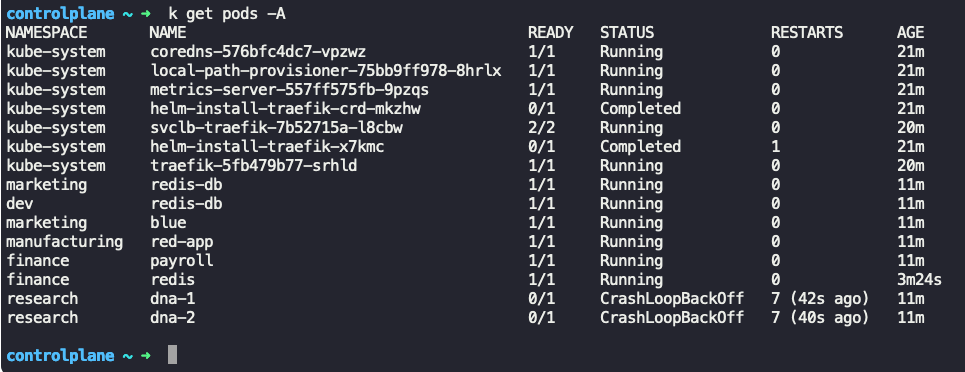


Create a pod in namespace=finance







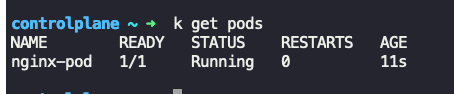


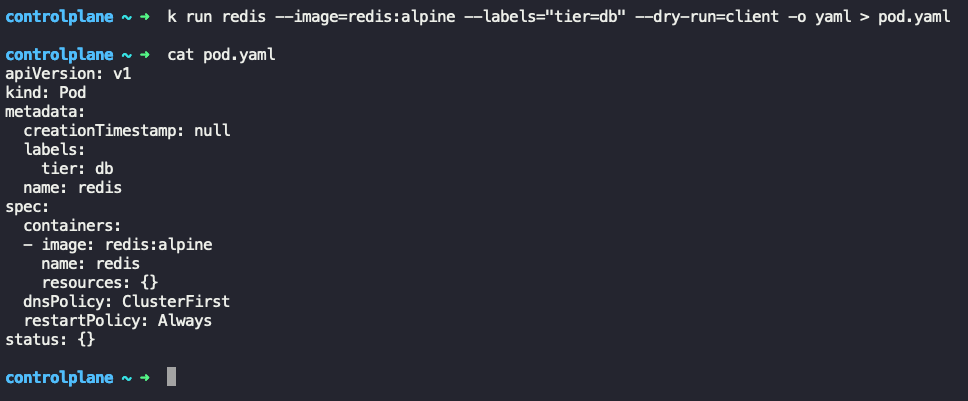
## [Imperative Commands]: Commands & Outputs



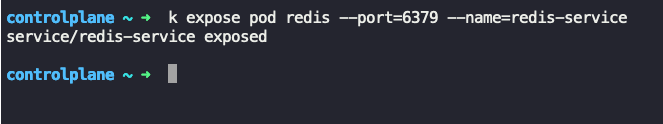








Expose a application(pod) named redis to port 6379 on service redis-service







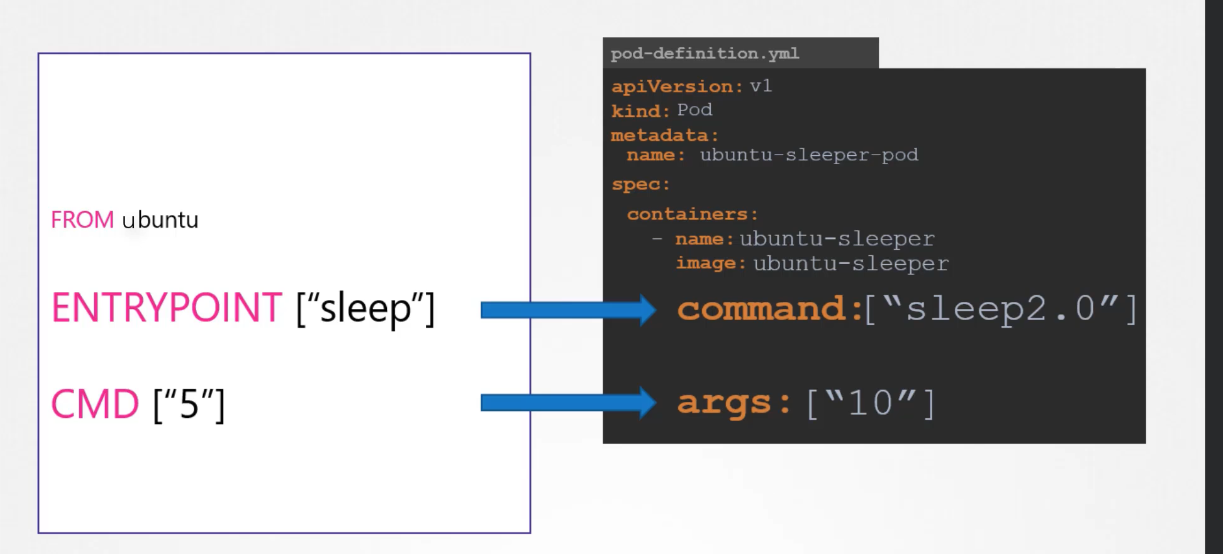


Create a deployment (redis-deploy), image(redis), replicas(2) and namespace(dev-ns)



Create a pod (httpd), image(httpd:alpine)  
--expose=True 🡪 Create a service of type ClusterIp and it requires –port option   
which is, port(80)

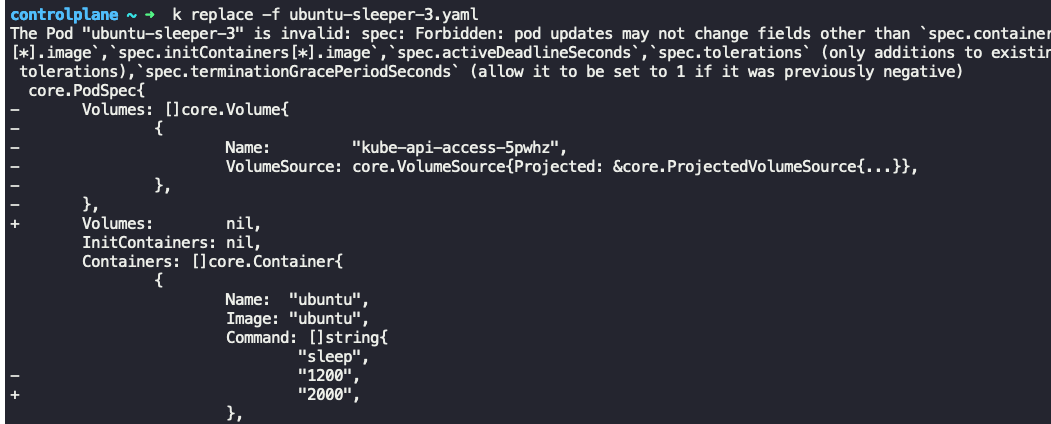




The **ENTRYPOINT** in the Dockerfile is overridden by the **command** in the pod definition file.

The **CMD** in the Dockerfile is overridden by the **args** in the pod definition file.

**Note**: It’s been noted that k edit command doesn’t work if you try to edit an exsiting command in above, it will complain and not allow to apply the modified changes.

Instead, edit the definition file, delete existing pod create the new pod using edited definition file  




# Annexure II

|  |  |  |
| --- | --- | --- |
| **Kind** | **apiVersion** | **Description** |
| **Pod** | v1 | The smallest and simplest Kubernetes object. Represents a single instance of a running process. |
| **ReplicaSet** | apps/v1 | Ensures that a specified number of pod replicas are running at any given time. |
| **Deployment** | apps/v1 | Manages ReplicaSets to ensure that the correct number of pods are running. |
| **StatefulSet** | apps/v1 | Manages the deployment and scaling of a set of Pods, providing guarantees about ordering and uniqueness. |
| **DaemonSet** | apps/v1 | Ensures that all (or some) nodes run a copy of a Pod. |
| **Job** | batch/v1 | Creates one or more pods and ensures that a specified number of them successfully terminate. |
| **CronJob** | batch/v1 | Manages time-based Job scheduling. |
| **Service** | v1 | Exposes a set of Pods as a network service. |
| **Ingress** | networking.k8s.io/v1 | Manages external access to services, usually HTTP. |
| **ConfigMap** | v1 | Provides configuration data to pods. |
| **Secret** | v1 | Stores sensitive information such as passwords, OAuth tokens, and SSH keys. |
| **PersistentVolume** | v1 | Represents a piece of storage in the cluster that has been provisioned by an administrator. |
| **PersistentVolumeClaim** | v1 | A request for storage by a user that dynamically provisions storage resources. |
| **Namespace** | v1 | Provides a mechanism for isolating groups of resources within a single cluster. |
| **Node** | v1 | Represents a worker node in Kubernetes. |
| **ServiceAccount** | v1 | Associates a Kubernetes Pod with a specific set of permissions. |
| **Role** | rbac.authorization.k8s.io/v1 | Grants access to resources within a single namespace. |
| **RoleBinding** | rbac.authorization.k8s.io/v1 | Associates a Role with a set of users or groups. |
| **ClusterRole** | rbac.authorization.k8s.io/v1 | Grants access to resources cluster-wide. |
| **ClusterRoleBinding** | rbac.authorization.k8s.io/v1 | Associates a ClusterRole with a set of users or groups across the entire cluster. |
| **NetworkPolicy** | networking.k8s.io/v1 | Controls the traffic flow to/from Kubernetes Pods. |
| **ResourceQuota** | v1 | Limits the amount of resources that a namespace can use. |
| **LimitRange** | v1 | Specifies limits to the resource usage for containers within a namespace. |
| **HorizontalPodAutoscaler** | autoscaling/v1 | Automatically scales the number of pods based on observed CPU utilization or other application metrics. |
| **EndpointSlice** | discovery.k8s.io/v1 | A scalable API for discovering network endpoints. |

**Notes:**

* **v1 API Group**: This is the core API group and contains most of the fundamental Kubernetes objects like Pods, Services, ConfigMaps, etc.
* **apps/v1 API Group**: This API group manages higher-level abstractions like Deployments, StatefulSets, ReplicaSets, and DaemonSets.
* **batch/v1 API Group**: This API group is used for batch jobs, including Job and CronJob.
* **networking.k8s.io/v1**: This API group handles network-related resources, such as Ingress and NetworkPolicy.
* **rbac.authorization.k8s.io/v1**: This API group is used for Role-Based Access Control (RBAC) resources like Role, RoleBinding, ClusterRole, and ClusterRoleBinding.
* **autoscaling/v1**: This API group is used for auto-scaling resources like the HorizontalPodAutoscaler.