Kubernetes Document

Kubernetes:

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. *K8s is developed by Google*.

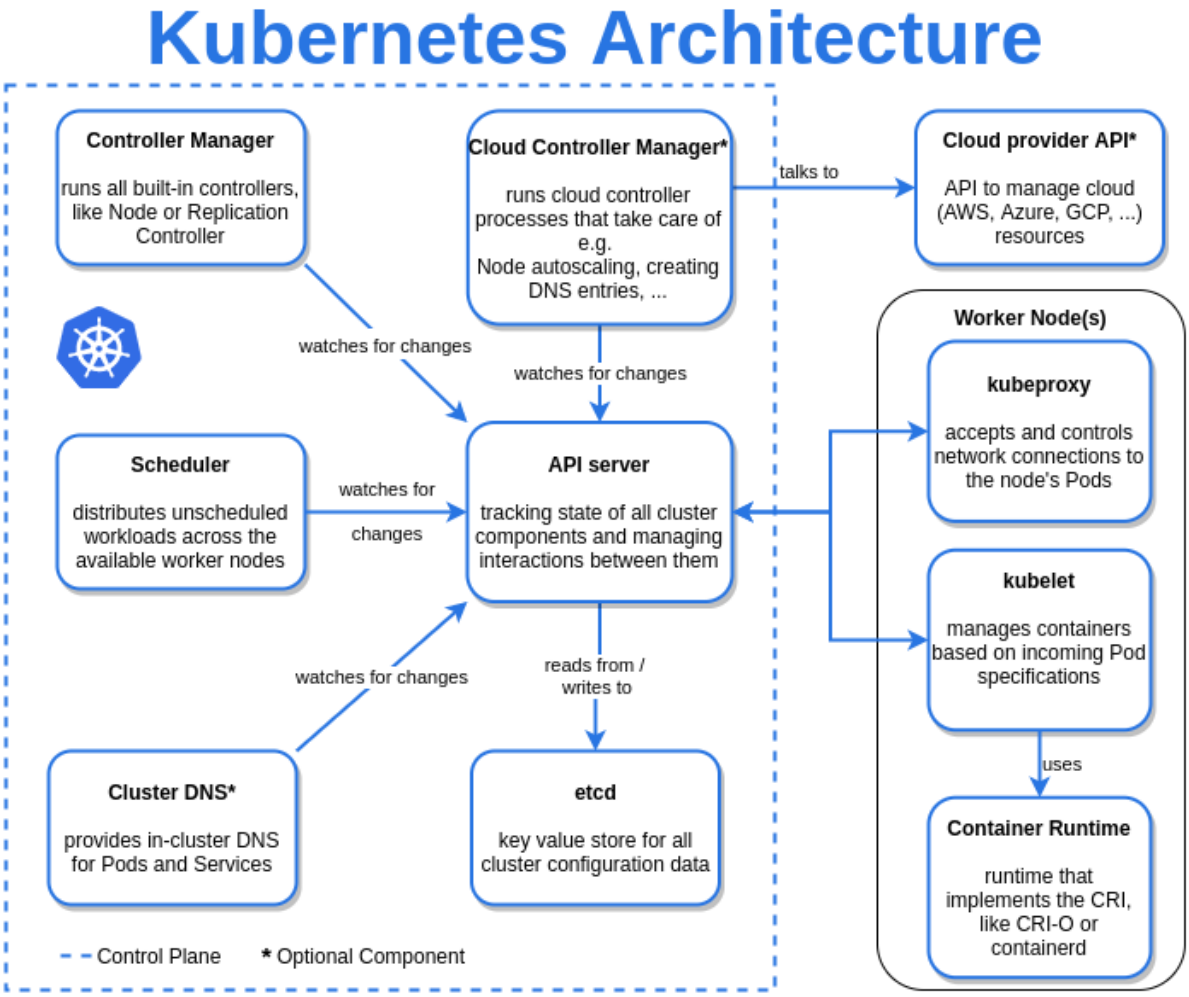
- **K8 is a Open Source, Container Orchestration Framework/tool.**
- **Means, K8s used to manage the applications made up by hundreds/thousands of containers in different deployment environments like physical machines, cloud machines etc.**

Need of Container Orchestration tool:

- **Usage of microservices increased the possibility of container technologies. Because containerization is the best way to host the small, independent services known as microservices.**
- **Now the rise of containers usage due to the microservice architecture results that each application depends on hundreds/thousands of containers.**

Managing these high amounts of containers in different environments using script/self made tools is not easy anymore. This actually causes the real need for the orchestration tool that manages the containers of the application. Managing here means giving High Availability, Scalability and Disaster recovery to the applications.

Kubernetes Components in Architecture:



Main components of Master Processes:

Master nodes are completely different processes running inside, which controls and manages a set of worker nodes. 4 processes should run in every master node to control the cluster state and worker nodes.

1.API Server:

- **It is the Frontend of the cluster. We use it for rest operation and connect to the api server. It will directly talk to etcd .**
- **When we deploy a new application in the K8s Cluster, We interact with the API Server through UI, Command Line tool or K8s API. This is like a gateway to the cluster to do all the changes in the cluster configurations.**

- **Also acts as an Authentication gatekeeper to allow only authenticated requests to the cluster. This is the only entry point to the K8s Cluster.**

2.Scheduler:

- **It is the one which schedules pods on specific node the pods want to run.It will schedule the pods based on label, taints, toleration.**
- **When we give a request to schedule a new Pod, the API Server will validate the request and send the request to the Scheduler. Then Scheduler will start the Pod in one of the worker nodes. Scheduler will see the resources requested for the application Pod, have the intelligence to assign it to the best worker node based on the resources available.**
- **The Scheduler will decide which node the application should go to, then Kubelet is the process that will start the container Pod with the specifications given by the Scheduler.**

3.Controller Manager:

- **It will manage the current state of the cluster.**
- **Controller manager is responsible for re-schedule of the applications when a crash happens. It detects the Cluster state changes. For example, if a pod dies, controller manager will find this as soon as possible and send the request to the Scheduler to reschedule the specific Pod. Then the Scheduler will do the request to Kubelet to schedule the Pod.**

4. etcd:

- **It is a key value store to the cluster state or database.It will keep all current state.**
- **All the changes happening in the cluster like new pod creation, Pod crash, Pod re-scheduling will be updated in this store. All the above three processes API Server, Scheduler and Controller-Manager will use this key value store data to know the current state of the Cluster. So this etcd component can be called as Cluster Brain which has all the data about the K8s cluster. This store only has cluster state data but not the application's data which is running in the Pods.**
- **Now, we see important master processes and how crucial these especially etcd data. This data must be reliably stored means replicated. In practice, K8s cluster is made up of more than one master node. All the master nodes run its own processes where etcd data store is distributed across the master nodes.**

Main components of K8s architecture of worker nodes:-

Each node can have one or more application pods where containers will be placed. Nodes are the servers which actually do the work.

Three important processes must be installed in every nodes to manage Pods.

1.Container-Runtime:

- **It takes care of actually running containers.**
- **It is a software that runs and manages the components required to run containers. This tool makes it easier to securely execute and efficiently deploy containers, a key component of container management.**

2.Kubelet:

- **It will pass the request to the container engine.**
- **Kubelet, responsible for communication between the Kubernetes Master and the Node, manages the Pods and the containers running on a machine.**
- **It is the primary "node agent" that runs on each node. It is actually registering the node server to the K8s. Kubelets working with the configuration called PodSpec. PodsSpec is a Yaml or Json object which actually describes the Pod with all the details. Kubelet ensures that all the containers described in the PodSpec are running fine and healthy by interacting with nodes and containers, and also takes care of assigning resources to the containers.**
- **The communication between the Pods handled by the Service component which is actually load balances and routes the request to the specific Pods.**

3.kube-proxy:

- **It will use Ip tables to provide an interface to connect kubernetes components.**
- **The actual process responsible for forwarding requests from Service components to Pods is Kube-Proxy which should be installed in all the nodes. It also makes sure that communication between Service and Pods is performant with low overhead.**

Components of Kubernetes:

1. Nodes:

- **Pods run on a Node. Node is a worker machine for either virtual/physical servers. A node can have multiple Pods. K8s master automatically handles scheduling the pods across the Nodes in the cluster.**

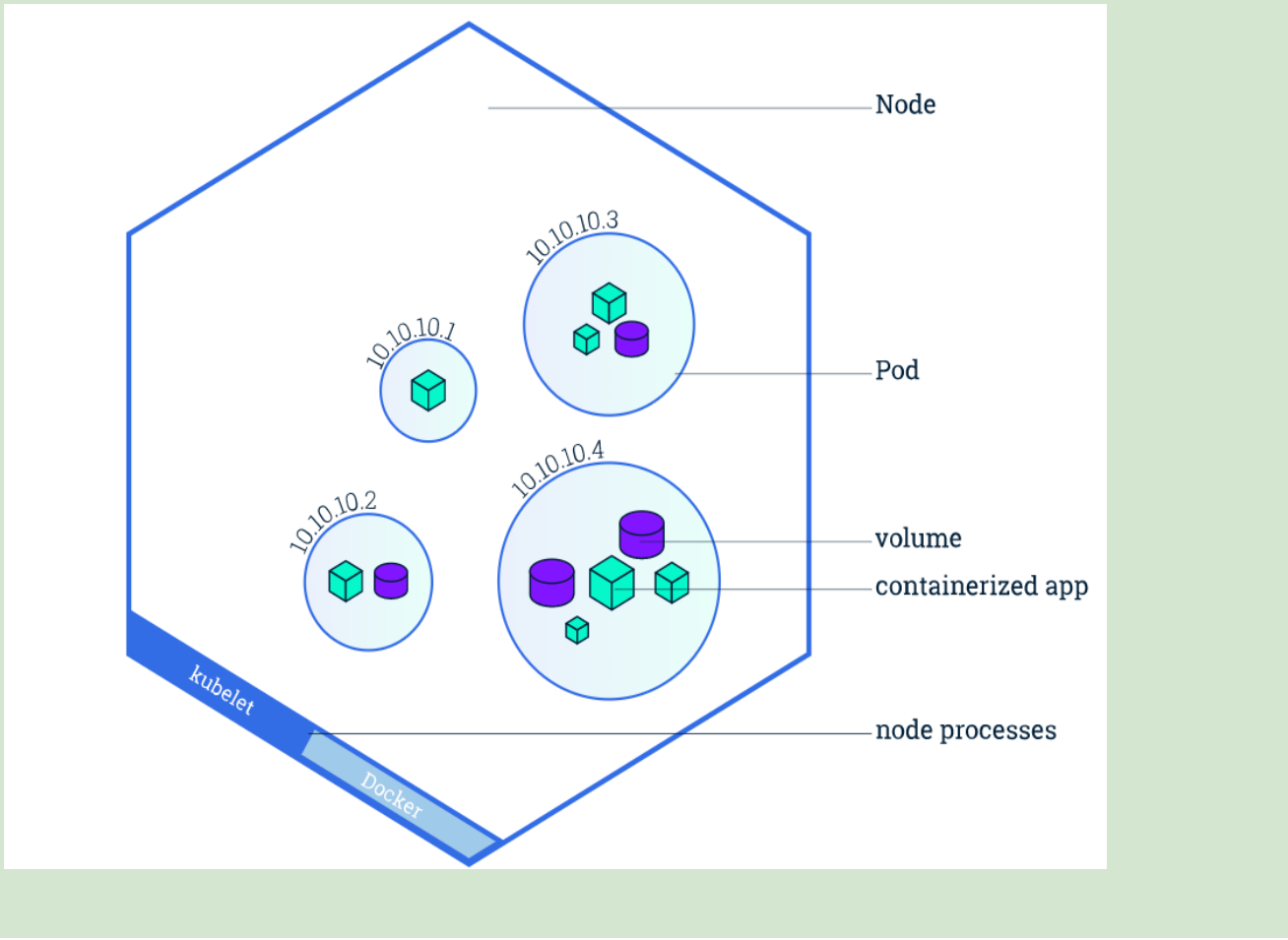


Image from official website

As seen in the above image, Node consists of many Pods. Pods have its own internal IP address which will be used to communicate between each other. Every Pods have their own storage and container images to be run. These IP addresses will be static/permanent. When a specific Pod is not running, A new pod will be created with the same IP so that the connection configuration no need to be changed.

- **Kubelet, responsible for communication between the Kubernetes Master and the Node; it manages the Pods and the containers running on a machine.**
- **There is a components named Service & Ingress plays important role. *Service used to give static IPs to the Pods*. All the requests from the**

outside world to your application domain will be coming to Ingest first. Then this Component will forward the requests to the specific requested Pod.

2. Pods:

- Pods are the smallest/basic unit you can create and manage in K8s. Pod is group of one or more containers with shared storage and network resources, have a specification on how to run the containers.
- Pods are something like a abstraction layer on top of the containers, creates the running environment for them. The abstraction is enforced by K8s because we only need to interact with Its own layers but not with the actual containers. Means that, Existing container images in the Kubernetes can be replaced by any other technology containers easily.
- Usually one container per Pod is the default scenario unless when the main application needs some small services(tightly coupled) which can be run in the same Pods.

3.Config Map & Secret:

Pods can connect to each other using Service feature. To make this connection, logically there should be somewhere the configuration needs to be stored right ? The man is named as configMap.

- Config Map is an Object used to store non-confidential data as a key-value pair.
- Pods use this Map to access each other. This config map can be used as environmental variables, command-line arguments or configuration files.
- This config Map allows you to decouple the configuration from your containers to make sure the containers are portable. For example, in your application, you need to replace your Sqlite3 database with a Mongo-db database. For this change, you just need to change in the ConfigMap and place your container in the Node and nothing else.
- Placing secret information like database connection details in ConfigMap is not advisable because this is just a plain text, no encryption applied to the data. For storing this, we can use Secret component.

4. Volume:

The data our application generates should be persistent, meaning when the container gets restarted, data should not be lost. Volumes is the component to achieve that. This can store the data in a local machine, meaning the node where the Pod is running or to the remote storage outside the K8s cluster (both cloud storage or own storage cluster).

5.Deployments:

Deployments are the blue-prints for the Pods. We create deployments to give updates regarding Pods and replicas. We know that Pods are the abstraction layer for the container images. Deployments are kind of abstraction layers for the Pods. So in practice, most of the time we will work with Deployments not with Pods. With this Deployment, we can replicate the application Pods but not the Database Pods. Because the database has state means data. if we replicate the DB Pod, inconsistency between the replicas may occur. StatefulSet is another component is K8s for this purpose.

6. StatefulSet:

So all the database applications should be created using StatefulSet but not with Deployments. Statefulset will take care the database is synchronized with the service and its replicas. But, doing this is not so easy in K8s. Common practice is running the database application outside the K8 cluster.

What if my application Pod dies?
Lets take my application Pod is dead and my user accessing it, it shows site cant reachable which is not good thing. To make sure this is not happening, everything in K8s is replicated. Means that, the application pod will be replicated to the another Node based on the configuration we provide. All these replicated Pods connected to the same service(remember IP is permanent). So replicated Pods also can be accessed with the same IP. Service component also does the Load balancing with these replicas.

##########################important parts in pod of kubernetes#################

1.securityContext:

$kubectl explain pods.spec.securityContext

KIND: Pod

VERSION: v1

RESOURCE: securityContext <Object>

DESCRIPTION:

#SecurityContext holds pod-level security attributes and common container settings. Optional: Defaults to empty. See type description for default values of each field.

#PodSecurityContext holds pod-level security attributes and common container settings. Some fields are also present in container.securityContext. Field values of container.securityContext take precedence over field values of PodSecurityContext.

2.jobs:

#To run the pod for specific time and stop the pod or certain time only pods will run

$kubectl explain jobs

KIND: Job

VERSION: batch/v1

DESCRIPTION:

Job represents the configuration of a single job.

3.cronjobs:

#To run cron jobs repeatedly

$kubectl explain cronjob.spec.schedule

KIND: CronJob

VERSION: batch/v1

FIELD: schedule <string>

DESCRIPTION:

The schedule in Cron format ("*/2 * * * *")

$kubectl get cron jobs

4.strategy(deployment specification):

#It will explain how pods should be updated.

5.selector(deployment specification):

#It uses match labels to identify how labels are matched against pods.

$labels:

#labels nothing but key value pairs.

6.template(deployment specification):

#We are given container information.

7.services:

ports:

 # Three types of ports for a service

1.nodePort - a static port assigned on each the node

2.port - port exposed internally in the cluster

3.targetPort - the container port to send requests to

########What does ClusterIP, NodePort, and LoadBalancer mean?###########

#The type property in the Service's spec determines how the service is exposed to the network.

#Kubernetes Service Types

1.ClusterIP

2.NodePort

3.LoadBalancer

#ClusterIP – The default value. The service is only accessible from within the Kubernetes cluster – you can't make requests to your Pods from outside the cluster!

#NodePort – This makes the service accessible on a static port on each Node in the cluster. This means that the service can handle requests that originate from outside the cluster.

#LoadBalancer – The service becomes accessible externally through a cloud provider's load balancer functionality.

-GCP, AWS, Azure, and OpenStack offer this functionality.

-The cloud provider will create a load balancer, which then automatically routes requests to your Kubernetes Service

8.readinessprobe:

It will check service or something is available or not in file before creating a pod

9.livenessprobe:

It will check if the service or something is available or not for every time based on time mentioned in the yml file while creating after running a pod.

1.http get -range 200-300 means good or above 400-500 means error

2.tcp socket- It will check if the tcp socket port is available or not in the pod - range 200-300 means good or above 400-500 means error.

10.taints:

Add a Taint on node to restrict pods from being scheduled

$kubectl taint nodes node1 app=red: NoSchedule

#The effects are as below;

1.NoSchedule — Pods will not be schedule on the nodes

2.PreferNoSchedule — The system will try to avoid placing a pod on the node, but it's not guaranteed

3.NoExecute — New pods will not be scheduled on the node and existing pods on the node if any will be evicted if they do not tolerate the taint

11.tolerations:

Add toleration on pod to allow specfic pod to be scheduled on the tainted node also.

12.Nodeselector:

The node selector is one such mechanism or constraint which we can apply to our pod to ensure that it is placed into a particular type of Node.

One has to specify the field

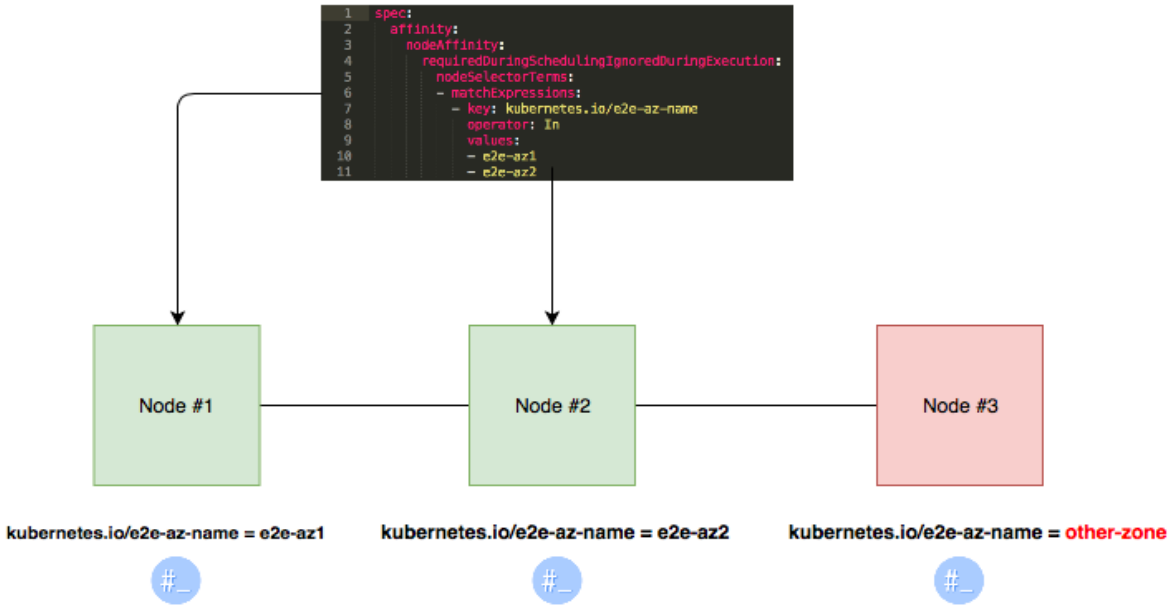
"nodeSelector",

which comes under the Pod's Spec definition. This field makes use of key-value pairs.

Any pod to be eligible for running onto any specific Node has to have the specified key-value pairs as labels of the intended Node.

13.NodeAffinity:

Node affinity allows scheduling Pods to specific nodes.



14.PodAffinity:

Pod Affinity ensures two pods to be co-located in a single node.

##################Kubectl command controlling the kubernetes cluster#############

#Kubectl---commands are used to interact and manage Kubernetes objects and the cluster.

######In this chapter, discuss commands used in Kubernetes via kubectl######

#################################kubectl cluster-info#############################

#known cluster info

$kubectl cluster-info

$kubectl cluster-info dump

$kubectl cluster-info dump --output-directory =/path/to/cluster-state

$kubectl config current-context

##############################kubectl get#####################################

#kubectl get – This command is capable of fetching data on the cluster about the Kubernetes resources.

$ kubectl get [(-o|--output=)json|yaml|wide|custom-columns=...|custom-columnsfile=...|go-template=...|go-template-file=...|jsonpath=...|jsonpath-file=...](TYPE [NAME | -l label] | TYPE/NAME ...) [flags]

For example,

 # List all pods in ps output format

$kubectl get pods

 # List all pods in ps output format with more information (such as node name)

$kubectl get pods -o wide

 # list all labels in the pod

$kubectl get pods --show-labels

#To Know pods in all the namespaces

```
$kubectl get pods  --all-namespaces

#copy the yaml file of already create
 $kubectl get deploy nginxserver -o  yaml > deploy-nginxserver.yml

#To Know all servicename
$kubectl get servicename or svc #svc short cut of servicename


# List a single replication controller with specified NAME in ps output format
$kubectl get replicationcontroller web

# List deployments in JSON output format, in the "v1" version of the "apps" API group
$kubectl get deployments.v1.apps -o json

# List a single pod in JSON output format
$kubectl get -o json pod web-pod-13je7

# List a pod identified by type and name specified in "pod.yaml" in JSON output format
$kubectl get -f pod.yaml -o  json

# List resources from a directory with kustomization.yaml - e.g. dir/kustomization.yaml
$kubectl get  -k dir/
# Return only the phase value of the specified pod
$kubectl get -o template pod/web-pod-13je7 --template={{.status.phase}}

# List resource information in custom columns
kubectl get pod test-pod -o
custom-columns=CONTAINER:.spec.containers[0].name,IMAGE:.spec.containers[0].image
```

 # List all replication controllers and services together in ps output format

$kubectl get rc,services

 # List one or more resources by their type and names

$kubectl get rc/web service/frontend pods/web-pod-13je7

############################kubectl logs##################################

#kubectl logs – They are used to get the logs of the container in a pod. Printing the logs can be defining the container name in the pod. If the POD has only one container there is no need to define its name.

$ kubectl logs [-f] [-p] POD [-c CONTAINER]

Examples:

 # Return snapshot logs from pod nginx with only one container

$kubectl logs nginx

 # Return snapshot logs for the pods defined by label app=nginx

$kubectl logs -lapp=nginx

 # Return snapshot of previous terminated ruby container logs from pod web-1

$kubectl logs -p -c ruby web-1

 # Begin streaming the logs of the ruby container in pod web-1

$kubectl logs -f -c ruby web-1

 # Display only the most recent 20 lines of output in pod nginx

$kubectl logs --tail=20 nginx

 # Show all logs from pod nginx written in the last hour

$kubectl logs --since=1h nginx

 # Return snapshot logs from first container of a job named hello

$kubectl logs job/hello

 # Return snapshot logs from container nginx-1 of a deployment named nginx

$kubectl logs deployment/nginx -c nginx-1

############################kubectl port-forward############################

#kubectl port-forward – They are used to forward one or more local port to pods.

$ kubectl port-forward POD [LOCAL_PORT:]REMOTE_PORT [...[LOCAL_PORT_N:]REMOTE_PORT_N]

Examples:

 # Listen on ports 5000 and 6000 locally, forwarding data to/from ports 5000 and 6000 in the pod

$kubectl port-forward pod/mypod 5000:6000

 # Listen on ports 5000 and 6000 locally, forwarding data to/from ports 5000 and 6000 in a pod selected by the deployment

$kubectl port-forward deployment/mydeployment 5000:6000

 # Listen on port 8443 locally, forwarding to the targetPort of the service's port named "https" in a pod selected by the service

$kubectl port-forward service/myservice 8443:https

 # Listen on port 8888 locally, forwarding to 5000 in the pod

$kubectl port-forward pod/mypod 8888:5000

 # Listen on port 8888 on all addresses, forwarding to 5000 in the pod

$kubectl port-forward --address 0.0.0.0 pod/mypod 8888:5000

 # Listen on port 8888 on localhost and selected IP, forwarding to 5000 in the pod

$kubectl port-forward --address localhost,10.19.21.23 pod/mypod 8888:5000

 # Listen on a random port locally, forwarding to 5000 in the pod

$kubectl port-forward pod/mypod :5000

##################################kubectl api-version#########################

#kubectl api-versions –It prints the supported versions of API on the cluster.

$#kubectl api-versions –It prints the supported versions of API on the cluster.

$ kubectl api-versions

admissionregistration.k8s.io/v1beta1

apiextensions.k8s.io/v1beta1

apiregistration.k8s.io/v1beta1

apps/v1

apps/v1beta1

apps/v1beta2

authentication.k8s.io/v1

authentication.k8s.io/v1beta1

authorization.k8s.io/v1

authorization.k8s.io/v1beta1

autoscaling/v1

autoscaling/v2beta1

batch/v1

batch/v1beta1

certificates.k8s.io/v1beta1

events.k8s.io/v1beta1

extensions/v1beta1

networking.k8s.io/v1

policy/v1beta1

rbac.authorization.k8s.io/v1

rbac.authorization.k8s.io/v1beta1

settings.k8s.io/v1alpha1

storage.k8s.io/v1

storage.k8s.io/v1beta1

v1

##################################kubectl run################################

#kubectl run – Run command has the capability to run an image on the Kubernetes cluster.

Creates a deployment or job to manage the created container(s).

$ kubectl run NAME --image=image [--env = "key = value"] [--port=port] [--replicas=replicas] [--dry-run=bool] [--overrides=inline-json] [--command] --[COMMAND] [args...]

Examples:

Start a single instance of nginx.

$kubectl run nginx --image=nginx

Start a single instance of hazelcast and let the container expose port 5701 .

$kubectl run hazelcast --image=hazelcast --port=5701

Start a single instance of hazelcast and set environment variables "DNS_DOMAIN=cluster" and "POD_NAMESPACE=default"

in the container.

$kubectl run hazelcast --image=hazelcast --env="DNS_DOMAIN=cluster" --env="POD_NAMESPACE=default"

Start a single instance of hazelcast and set labels "app=hazelcast" and "env=prod" in the container.

$kubectl run hazelcast --image=nginx --labels="app=hazelcast,env=prod"

Start a replicated instance of nginx.

kubectl run nginx --image=nginx --replicas=5

Dry run. Print the corresponding API objects without creating them.

$kubectl run nginx --image=nginx --dry-run

Start a single instance of nginx, but overload the spec of the deployment with a partial set of values parsed from JSON.

$kubectl run nginx --image=nginx --overrides='{ "apiVersion": "v1", "spec": { ... } }'

Start a pod of busybox and keep it in the foreground, don't restart it if it exits.

$kubectl run -i -t busybox --image=busybox --restart=Never

Start the nginx container using the default command, but use custom arguments (arg1 .. argN) for that command.

$kubectl run nginx --image=nginx -- <arg1> <arg2> ... <argN>

Start the nginx container using a different command and custom arguments.

$kubectl run nginx --image=nginx --command -- <cmd> <arg1> ... <argN>

Start the perl container to compute π to 2000 places and print it out.

$kubectl run pi --image=perl --restart=OnFailure -- perl -Mbignum=bpi -wle 'print bpi(2000)'

Start the cron job to compute π to 2000 places and print it out every 5 minutes.

$kubectl run pi --schedule="0/5 * * * ?" --image=perl --restart=OnFailure -- perl -Mbignum=bpi -wle 'print bpi(2000)'

##############################kubectl create##############################

#kubectl create – To create resources by filename of or stdin. To do this, JSON or YAML formats are accepted.

$$kubectl create --help | less

Examples:

 # Create a pod using the data in pod.json

$ kubectl create –f <File Name>

$kubectl create -f ./pod.json

$kubectl create deployment nginx --image=nginx

$kubectl create service nodeport nginx --tcp=80:80

 # Create a pod based on the JSON passed into stdin

cat pod.json | kubectl create -f -

 # Edit the data in docker-registry.yaml in JSON then create the resource using the edited data

$kubectl create -f docker-registry.yaml --edit -o json

#create deployment pods and displays yaml file

$ kubectl create deployment nginx server --image=nginx -o yaml

Available Commands:

clusterrole Create a cluster role

clusterrolebinding Create a cluster role binding for a particular cluster role

configmap Create a config map from a local file, directory or literal value

cronjob Create a cron job with the specified name

deployment Create a deployment with the specified name

ingress Create an ingress with the specified name

job Create a job with the specified name

namespace Create a namespace with the specified name

poddisruptionbudget Create a pod disruption budget with the specified name

priorityclass Create a priority class with the specified name

quota Create a quota with the specified name

role Create a role with single rule

rolebinding Create a role binding for a particular role or cluster role

secret Create a secret using specified subcommand

service Create a service using a specified subcommand

serviceaccount Create a service account with the specified name

##########kubectl replace##########

#kubectl replace – Capable of replacings new modified in file then we use replace for yml file

$kubectl replace --help | less

Examples:

Replace a pod using the data in pod.json.

$kubectl replace -f ./pod.json

Replace a pod based on the JSON passed into stdin.

$cat pod.json | kubectl replace -f -

Update a single-container pod's image version (tag) to v4

$kubectl get pod mypod -o yaml | sed 's/\(image: myimage\):.*$/\1:v4/' | kubectl replace -f -

Force replace, delete and then re-create the resource

$kubectl replace --force -f ./pod.json

##########kubectl apply##########

#kubectl apply – It has the capability to configure a resource by file or stdin or new create and for replace new conatiner in pod.

Examples:

Apply the configuration in pod.json to a pod.

$kubectl apply -f ./pod.json

Apply the JSON passed into stdin to a pod.

$ cat pod.json | kubectl apply -f -

 # Note: --prune is still in Alpha

 # Apply the configuration in manifest.yaml that matches label app=nginx and delete all the other resources that are not in the file and match label app=nginx.

$kubectl apply --prune -f manifest.yaml -l app=nginx

 # Apply the configuration in manifest.yaml and delete all the other configmaps that are not in the file.

$kubectl apply --prune -f manifest.yaml --all --prune-whitelist=core/v1/ConfigMap.

##############################kubectl attach##############################

#kubectl attach – Attach to a process that is already running inside an existing container.

Examples:

 # Get output from running pod 123456-7890, using the first container by default

$kubectl attach 123456-7890

 # Get output from ruby-container from pod 123456-7890

$kubectl attach 123456-7890 -c ruby-container

 # Switch to raw terminal mode, sends stdin to 'bash' in ruby-container from pod 123456-7890

 # and sends stdout/stderr from 'bash' back to the client

$kubectl attach 123456-7890 -c ruby-container -i -t

 # Get output from the first pod of a ReplicaSet named nginx

$kubectl attach rs/nginx

##############################kubectl delete##############################

#kubectl delete – Deletes resources by file name, stdin, resource and names.

$kubectl delete --help | less

Examples:

 # Delete a pod using the type and name specified in pod.json.

$kubectl delete -f ./pod.json

 # Delete a pod based on the type and name in the JSON passed into stdin.

$cat pod.json | kubectl delete -f -

 # Delete pods and services with same names "baz" and "foo"

$kubectl delete pod,service baz foo

 # Delete pods and services with label name=myLabel.

$kubectl delete pods,services -l name=myLabel

 # Delete a pod with minimal delay

$kubectl delete pod foo --now

 # Force delete a pod on a dead node

$kubectl delete pod foo --grace-period=0 --force

 # Delete all pods

$kubectl delete pods --all

##############################kubectl describe##################################

#kubectl describe – Describes any particular resource in kubernetes. Shows details of a resource or a group of resources.

#Print a detailed description of the selected resources, including related resources such as events or controllers. You

#may select a single object by name, all objects of that type, provide a name prefix, or label selector. For example:

$ kubectl describe TYPE NAME_PREFIX

#will first check for an exact match on TYPE and NAME PREFIX. If no such resource exists, it will output details for

#every resource that has a name prefixed with NAME PREFIX.

#Valid resource types include:

*** all**

*** certificatesigningrequests (aka 'csr')**

*** clusterrolebindings**

*** clusterroles**

*** componentstatuses (aka 'cs')**

*** configmaps (aka 'cm')**

*** controllerrevisions**

*** cronjobs**

*** customresourcedefinition (aka 'crd')**

*** daemonsets (aka 'ds')**

*** deployments (aka 'deploy')**

*** endpoints (aka 'ep')**

*** events (aka 'ev')**

*** horizontalpodautoscalers (aka 'hpa')**

*** ingresses (aka 'ing')**

*** jobs**

*** limitranges (aka 'limits')**

*** namespaces (aka 'ns')**

*** networkpolicies (aka 'netpol')**

*** nodes (aka 'no')**

*** persistentvolumeclaims (aka 'pvc')**

*** persistentvolumes (aka 'pv')**

*** poddisruptionbudgets (aka 'pdb')**

*** podpreset**

* pods (aka 'po')
* podsecuritypolicies (aka 'psp')
* podtemplates
* replicasets (aka 'rs')
* replicationcontrollers (aka 'rc')
* resourcequotas (aka 'quota')
* rolebindings
* roles
* secrets
* serviceaccounts (aka 'sa')
* services (aka 'svc')
* statefulsets (aka 'sts')
* storageclasses (aka 'sc')

Examples:

```
# Describe a node
$kubectl describe nodes kubernetes-node-emt8.c.myproject.internal

# Describe a pod
$kubectl describe pods/nginx

# Describe a pod identified by type and name in "pod.json"
$kubectl describe -f pod.json

# Describe all pods
$kubectl describe pods

# Describe pods by label name=myLabel
```

$kubectl describe po -l name=myLabel

Describe all pods managed by the 'frontend' replication controller (rc-created pods

get the name of the rc as a prefix in the pod the name).

$kubectl describe pods frontend

############################kubectl drain#################################

#kubectl drain – This is used to drain a node for maintenance purpose. It prepares the node for maintenance. This will mark the node as unavailable so that it should not be assigned with a new container which will be created.

Examples:

Drain node "foo", even if there are pods not managed by a ReplicationController, ReplicaSet, Job, DaemonSet or StatefulSet on it.

$ kubectl drain foo --force

############################kubectl edit#################################

#kubectl edit – It is used to end the resources on the server. This allows to directly edit a resource which one can receive via the command line tool.

$ kubectl edit <Resource/Name | File Name)

Examples:

Edit the service named 'docker-registry':

$kubectl edit svc/docker-registry

Use an alternative editor

$KUBE_EDITOR="nano" kubectl edit svc/docker-registry

Edit the job 'myjob' in JSON using the v1 API format:

$kubectl edit job.v1.batch/myjob -o json

Edit the deployment 'mydeployment' in YAML and save the modified config in its annotation:

$kubectl edit deployment/mydeployment -o yaml --save-config

####################################kubectl exec#########################

#kubectl exec – This helps to execute a command in the container.

Examples:

 # Get output from running 'date' from pod 123456-7890, using the first container by default

$kubectl exec 123456-7890 date

 # Get output from running 'date' in ruby-container from pod 123456-7890

$kubectl exec 123456-7890 -c ruby-container date

 # Switch to raw terminal mode, sends stdin to 'bash' in ruby-container from pod 123456-7890

 # and sends stdout/stderr from 'bash' back to the client

$kubectl exec 123456-7890 -c ruby-container -i -t -- bash -il

 # List contents of /usr from the first container of pod 123456-7890 and sort by modification time.

 # If the command you want to execute in the pod has any flags in common (e.g. -i),

 # you must use two dashes (--) to separate your command's flags/arguments.

 # Also note, do not surround your command and its flags/arguments with quotes

 # unless that is how you would execute it normally (i.e., do ls -t /usr, not "ls -t /usr").

$kubectl exec 123456-7890 -i -t -- ls -t /usr

############################kubectl expose#################################

#kubectl expose – This is used to expose the Kubernetes objects such as pod, replication controller, and service as a new Kubernetes service.

-This has the capability to expose it via a running container or from a yaml file.

$ kubectl expose (-f FILENAME | TYPE NAME) [--port=port] [--protocol = TCP|UDP] [--target-port = number-or-name] [--name = name] [--external-ip =external-ip-of service] [--type = type]

Possible resources include (case insensitive):

pod (po), service (svc), replicationcontroller (rc), deployment (deploy), replicaset (rs)

Examples:

 # Create a service for a replicated nginx, which serves on port 80 and connects to the containers on port 8000

$kubectl expose rc nginx --port=80 --target-port=8000

 # Create a service for a replication controller identified by type and name specified in "nginx-controller.yaml",

which serves on port 80 and connects to the containers on port 8000

$kubectl expose -f nginx-controller.yaml --port=80 --target-port=8000

 # Create a service for a pod valid-pod, which serves on port 444 with the name "frontend"

$kubectl expose pod valid-pod --port=444 --name=frontend

 # Create a second service based on the above service, exposing the container port 8443 as port 443 with the name "nginx-https"

$kubectl expose service nginx --port=443 --target-port=8443 --name=nginx-https

 # Create a service for a replicated streaming application on port 4100 balancing UDP traffic and named 'video-stream'.

$kubectl expose rc streamer --port=4100 --protocol=UDP --name=video-stream

 # Create a service for a replicated nginx using replica set, which serves on port 80 and connects to the containers on port 8000

$kubectl expose rs nginx --port=80 --target-port=8000

 # Create a service for an nginx deployment, which serves on port 80 and connects to the containers on port 8000

$kubectl expose deployment nginx --port=80 --target-port=8000

##################################kubectl autoscale ######################

#kubectl autoscale – This is used to auto scale pods which are defined such as Deployment, replica set, Replication Controller.

$ kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min = MINPODS] --max = MAXPODS [--cpu-percent = CPU] [flags]

$ kubectl autoscale deployment foo --min = 2 --max = 10

####################To switch to different namespaces ####################

#kubectl config set-context – Sets the current context in the kubectl file of the namespace to switch to a different namespace.

$ kubectl config set-context --current --namespace=default

#To use namespace easily we can download kubectx from git and clone the server

$ git clone https://github.com/ahmetb/kubectx.git

$ cp kubectx kubens /usr/local/bin/

############################kubectl config############################

#kubectl config view

$ kubectl config view

$ kubectl config view –o jsonpath='{.users[?(@.name == "e2e")].user.password}'

############################kubectl cp############################

#kubectl cp – Copy files and directories to and from containers.

Examples:

 # !!!Important Note!!!

 # Requires that the 'tar' binary is present in your container

 # image. If 'tar' is not present, 'kubectl cp' will fail.

 # Copy /tmp/foo_dir local directory to /tmp/bar_dir in a remote pod in the default namespace

 $kubectl cp /tmp/foo_dir <some-pod>:/tmp/bar_dir

 # Copy /tmp/foo local file to /tmp/bar in a remote pod in a specific container

 $kubectl cp /tmp/foo <some-pod>:/tmp/bar -c <specific-container>

 # Copy /tmp/foo local file to /tmp/bar in a remote pod in namespace <some-namespace>

$ kubectl cp /tmp/foo <some-namespace>/<some-pod>:/tmp/bar

Copy /tmp/foo from a remote pod to /tmp/bar locally

$kubectl cp <some-namespace>/<some-pod>:/tmp/foo /tmp/bar

Options:

-c, --container='': Container name. If omitted, the first container in the pod will be chosen.

############################kubectl rolling-update#########################

#kubectl rolling-update – Performs a rolling update on a replication controller.

-Replaces the specified replication controller with a new replication controller by updating a POD at a time.

$ kubectl rolling-update OLD_CONTROLLER_NAME ([NEW_CONTROLLER_NAME] --image = NEW_CONTAINER_IMAGE | -f NEW_CONTROLLER_SPEC)

$ kubectl rolling-update frontend-v1 –f freontend-v2.yaml

############################kubectl rollout#####################################

#kubectl rollout – It is capable of managing the rollout of deployment.

Manage the rollout of a resource.

Valid resource types include:

* deployments
* daemonsets
* statefulsets

Examples:

Rollback to the previous deployment

$kubectl rollout undo deployment/abc

Check the rollout status of a daemonset

$kubectl rollout status daemonset/foo

#Available Commands:

history View rollout history

pause Mark the provided resource as paused

restart Restart a resource

resume Resume a paused resource

status Show the status of the rollout

undo Undo a previous rollout

#Apart from the above, we can perform multiple tasks using the rollout such as –

rollout history

rollout pause

rollout resume

rollout status

rollout undo

#To check rollout history

$ kubectl rollout history deploy nginx

deployment.apps/nginx

REVISION CHANGE-CAUSE

2 <none>

3 <none>

#To rollout back previous versions

$kubectl rollout undo deploy nginx --to-revision=2

deployment.apps/nginx rolled back

##

strategy:

rollingUpdate:

maxSurge: 2 ###

maxUnavailable: 1 ##

type: RollingUpdate

##kubectl scale##########################

#kubectl scale – It will scale the size of Kubernetes Deployments, ReplicaSet, Replication Controller, or job.

$ kubectl scale [--resource-version = version] [--current-replicas = count] --replicas = COUNT (-f FILENAME | TYPE NAME)

#If --current-replicas or --resource-version is specified, it is validated before the scale is attempted, and it is

-guaranteed that the precondition holds true when the scale is sent to the server.

Examples:

 # Scale a replica set named 'foo' to 3

kubectl scale --replicas=3 rs/foo #rs=ReplicaSet

 # Scale a resource identified by type and name specified in "foo.yaml" to 3

kubectl scale --replicas=3 -f foo.yaml #foo.yaml=yaml file

 # If the deployment named mysql's current size is 2, scale mysql to 3

kubectl scale --current-replicas=2 --replicas=3 deployment/mysql

 # Scale multiple replication controllers

kubectl scale --replicas=5 rc/foo rc/bar rc/baz #rc=Replication Controller

 # Scale stateful set named 'web' to 3

kubectl scale --replicas=3 statefulset/web

############################kubectl set resources############################

#kubectl set resources – It is used to set the content of the resource. It updates resource/limits on objects with pod templates.

#Specify compute resource requirements (CPU, memory) for any resource that defines a pod template. If a pod is successfully scheduled, it is guaranteed the amount of resource requested, but may burst up to its specified limits.

#For each compute resource, if a limit is specified and a request is omitted, the request will default to the limit.

Possible resources include (case insensitive): Use "kubectl api-resources" for a complete list of supported resources..

Examples:

 # Set a deployments nginx container cpu limits to "200m" and memory to "512Mi"

$ kubectl set resources deployment nginx -c=nginx --limits=cpu=200m,memory=512Mi

 # Set the resource request and limits for all containers in nginx

$kubectl set resources deployment nginx --limits=cpu=200m,memory=512Mi --requests=cpu=100m,memory=256Mi

 # Remove the resource requests for resources on containers in nginx

$kubectl set resources deployment nginx --limits=cpu=0,memory=0 --requests=cpu=0,memory=0

 # Print the result (in yaml format) of updating nginx container limits from a local, without hitting the server

$ kubectl set resources -f path/to/file.yaml --limits=cpu=200m,memory=512Mi --local -o yaml

####################################kubectl top#############################

#kubectl top node or pods – It displays CPU/Memory/Storage usage.

#The top command allows you to see the resource consumption for nodes or pods.

#This command requires Heapster to be correctly configured and working on the server.

Available Commands:

node Display Resource (CPU/Memory/Storage) usage of nodes

pod Display Resource (CPU/Memory/Storage) usage of pods

Usage:

$kubectl top [options]

################################kubectl set image############################

###kubectl set image – Update existing container image(s) of resources..

$ kubectl set image (-f FILENAME | TYPE NAME) CONTAINER_NAME_1 = CONTAINER_IMAGE_1 ... CONTAINER_NAME_N = CONTAINER_IMAGE_N

#Possible resources include (case insensitive):

pod (po), replicationcontroller (rc), deployment (deploy), daemonset (ds), replicaset (rs)

Examples:

 # Set a deployment's nginx container image to 'nginx:1.9.1', and its busybox container image to 'busybox'.

$kubectl set image deployment/nginx busybox=busybox nginx=nginx:1.9.1

 # Update all deployments' and rc's nginx container's image to 'nginx:1.9.1'

$kubectl set image deployments,rc nginx=nginx:1.9.1 --all

 # Update image of all containers of daemonset abc to 'nginx:1.9.1'

$kubectl set image daemonset abc *=nginx:1.9.1

 # Print result (in yaml format) of updating nginx container image from local file, without hitting the server

$kubectl set image -f path/to/file.yaml nginx=nginx:1.9.1 --local -o yaml

Summarize:

- **Pod component is an abstraction layer for the container image(s) which runs in the Node(server). One or more Pods can run on the server.**
- **Service component is used to have the communication between the Pods.**
- **Ingress component is to route the traffic to the K8 cluster.**
- **Configuration for the Pods can be stored in the ConfigMap and Secrets.**
- **Data storage is happening using the Volumes component.**
- **Deployments & Statefulsets are for Pods replica configurations.**