* Container and orchestration
* Docker Image
  + Docker image is application template, where we can create and run the similar multiple applications from the Docker Image
* Container
  + Container is the running unit of Docker image.
* Main diff between hypervisors and containers
  + Hardware > OS > Hypervisor > here on top of it we have multiple VM OS > its dependencies>application
  + Hardware>OS>Docker>here on top of docker we can run multiple containers of same OS kernel>Debian, fedor, suse & centOS (it means that any application related to these OS can be hosted on docker and can be managed without having any multiple dependencies for every OS unlike hypervisor)
  + What if I am using OS with different kernel like Windows, for this case we need to deploy -> Hardware > OS > Docker > Windows containers need to be deployed, here we cannot deploy Containers that have different OS kernel.

**Container Orchestration**

Orchestration technologies we have is **Docker Swarm** from Docker, **Kubernetes** from Google, **MESOS** from Apache.

* Here Orchestration comes into picture when we need to mange these containers deployment, connectivity, High Availability, Upscale, downscale etc.
* We can just write Yaml script files to do all of these activities.

**Kubernetes Architecture**

* Here in Kubernetes architecture, we have below components,

1. Node
2. Master Node
3. Worker Node
4. Kubernetes Cluster

* **Node:**

Node is Virtual Machine or a physical Machine where containers are hosted

* **Kubernetes Cluster:**

Multiple Nodes put together called as Kubernetes Cluster

* **Master Node:**

One node is needed to manage Kubernetes Cluster is known as Master node

* **Worker Node:**

All the nodes created with in the cluster are known as Worker nodes.

We have multiple components that are related to Master node, worker node/minions, Kubernetes Cluster.

* **API Server** – all the commands we execute from user device will talk to API server of the Kubernetes cluster
* **Key-value store** – It is a distributed reliable key value store that stores all the information that needs to manage the Kubernetes cluster and also it stores all the master nodes and the worker nodes related information and also generates logs within the cluster.
* **Scheduler** – is responsible to share the work load of the application among the worker nodes(along containers) in the cluster
* **Controller** – This is brain behind the cluster and responsible, if any container or node goes down it tries to bring up node or containers.
* **Container Runtime** - it is piece of software used to run the image, in our case it is “Docker Engine”, we also have “Containerd”, “CRI-O”, “Rocket(rkt)” & “Mirantis Container Runtime(MCR)”
* **Kubelet** - This an agent present in each node and responsible for monitoring containers are running or not & healthy or not.
* **Kubectl** – This is command line utility tools used to manage Kubernetes cluster

**Above components are bifurcated between Master node and Worker node as follows,**

**Master Node**

* Kube- apiserver
* Etcd
* Controller
* Scheduler

**Worker Nodes**

* Kubelet
* Container Runtime

**Note:**

* Apiserver via kubelet performs all the actions like checking the nodes and containers health and performs all the actions needed.
* Etcd will store all the information collected.
* We also have other components, later will focus on that.

**PODs:**

* POD is the smallest object in Kubernetes where the containers are hosted by default,
* Same container docker image cannot be in a single POD, different docker image container can be in same POD
* Containers in the same POD have same network space and volume
* POD will be assigned with IP and the Node also will be assigned with IP

**PODs with YAML:+**

* Now we see writing yaml files for creating PODs, so here first we need to know the default structure of an Yaml file, below are the four common keys we will have

**Ex:** apiVersion: v1

Kind: POD

Metadata:

Name: myapp-pod

Lables:

App: myapp

Type: front-end

Spec:

Containers:

-- Name: nginx-container

Image: nginx

* From above default keys “apiVersion” depends upon the kind of resource we plan to deploy/provision, in our case it is for “POD”, we have different apiVersions for different resources as shown below

|  |  |
| --- | --- |
| **Kind** | **Version** |
| POD | v1 |
| Service | v1 |
| ReplicaSet | apps/v1 |
| Deployment | apps/v1 |

* And here we need to remember that under “metadata” dictionary “name & labels” keys are common and these cannot be changed, and under “lables” we can create our own labels which is needed like “app & type”
* And container will be specified under “Spec”

A screenshot of a computer program

Description automatically generated

* From above image “READY” column indicates “Running containers in a POD/total number of containers in POD ”

**Replication Controller/ ReplicaSet**

* Now we goanna discuss about the Replication controller feature, it is the brain behind the High availability of the application pods hosted on nodes, what it does is while creating Pods we tag the lables like : frontend/backend etc.., depending on that Replication Controller creates a logical grouping, for instance if the any one frontend application pod goes down, immediately it creates a new pod, by this it helps for the availability of our application all the time.
* And it has features like can set how many “replicas” we need in a logical group, it ensures to maintain that “desired” state, for example we set “replicas = 3” in our replication\_defination yaml file, and we are trying to deploy 4th replica into our logical group it wont allow us to create it deletes immediately
* Replication Controller has become old version and updated version of it called as “ReplicaSet”
* Only the difference it has, it can add pre-existing Pods automatically into ReplicaSet logical group when ReplicaSet created if needed.
* On the go we can can scale our Pods by editing our existing “ReplicaSet\_defenition.yaml” or we can edit by using this command **“kubectl edit ReplicaSet <ReplicaSetName>”** it will open running ReplicaSet definition.yaml file to edit.

Command to remember.

* Kubectl get pods
* Kubectl get nodes
* Kubectl get ReplicaSet
* Kubectl describe pod/<podname>
* Kubectl describe Node/<nodename>
* Kubectl describe ReplicaSet/<replicatesetname>
* Kubectl delete pod <podname>
* Kubectl delete node <nodename>
* Kubectl delete ReplicaSet <ReplicaSetname>
* Kubectl get pods -o wide
* Kubectl get nodes -o wide
* Kubectl get ReplicaSet -o wide

**Deployments -Update and Rollback:**

Commands:

* + Kubectl create -f deployment-definition.yml --record
  + Kubectl get deployments
  + Kubectl apply -f deployment-definition.yml --record(update)
  + Kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1 --record (Updating the existing pod application version)
  + Kubectl rollout status deployment/myapp-deployment
  + Kubectl rollout history deployment/myapp-deployment
  + Kubectl rollout undo deploy (rollback to previous state)

Here we come up with new Feature called “Deployments”

* Deployment definition file and Replicaset definition file are same only onething changes is “kind: Deployment”

Example:

apiVersion: apps/v1

kind: Deployment

metadata:

  name: frontend

  labels:

    app: mywebsite

    tier: frontend

spec:

  replicas: 4

  template:

    metadata:

      name: myapp-pod

      labels:

        app: myapp

    spec:

      containers:

        - name: nginx

          image: nginx

  selector:

    matchLabels:

        app: myapp

* In Deployments main feature is to update existing container application version, without impacting to the application.
* We have two types of Rolling updates Strategy one is “Rolling update”(this is default strategy) other is “Recreate”
* In Rolling Update strategy, it updates application one after the other container, if we have 3 replicas in a deployment it starts with one container brings down > upgradeit > bringup and then second container bring down > upgradeit > bringup and continues …
* In Recreate it brings down all the containers > upgradeall > bringup all the containers here we see application down time.
* And also we can roll back our upgrades if that doesn't work, mentioned commands above how can we achieve those by changing the “container image version” and by changing the “Strtegy” Rolling update or Recreate

**Kubernetes Basic Networking:**

* By default node is allocated with IP address and the Pod inside node also allocated with unique Ip address, here the IP address of the Pod is picked up from the pre existing network configured with in the node.
* Communication between multiple nodes and the pods can be done only by using popular network solutions “cilium, Flannel, VMware NSX & Cisco etc.”, which helps us to set up the network path between nodes with in the Cluster to flow the traffic.

**Services:**

* Now we talk about Kubernetes “Services”, it’s like one of the Kubernetes features we have ReplicaSet & Deployment.
* Here we use this service to make web applications accessible to external users

**NodePort Service:**

* In detail node have IP address > inside Pod will have IP address of different subnet by default node and Pod doesn’t have direct communication, usually for external user to access the web application hosted on pod need to ssh to node IP address and then try to curl the web application url to access, now exactly here we need to access the application directly instead of accessing node, in this use case we use “NodePort” Service to establish the connectivity between our node and pod
* So by creating service definition file with NodePort, Port, Target Port shown below illustration we can access our web application.

Example: consider below ports Service definition we have done, so that we can access the application “curl http://<nodeIPaddress>:<Nodeport>” i.e,. curl http://192.168.2.3:30008

A diagram of a computer

Description automatically generated

* Here we have three types in Kubernetes Service 1. NodePort, 2. ClusterIP, 3. Load Balancer, above illustration is example of NodePort Service type.

**Note:**

* NodePort Service use the “Algorithm: Random” and “SessionAffinity: Yes”
* And also every time the service is created Service itself will have one IP assigned called as “CluseterIP” and another IP for the Nodeport Service

**ClusterIP Service:**

* Here the ClusterIP service is used for 3 tier application environments, it establishes the communication between front end application to backend and for back end application to database servers, it creates a logical interface where all the frontend applications communicate through the logical interface established by the Cluster IP service with backend applications same with backend to data base servers

A diagram of a network

Description automatically generated

**LoadBalancer Service:**

* Here these Load balancer Service is similar to Node port service, where frontend application is accessible via nodeport, only the differnence is, if we have multiple Pods hosted on multiple nodes we can access the application using multiple node urls instead of having single url
* So in order to get a single url to access our front end applications we use this service (in service definition file under “Spec” type would be “Load balancer”), but it supports only cloud native load balancers like “GCP/AWS/Azure” rest everything is same

**For example:**

Lets consider above node port illustration and we have same environment but on multiple nodes, so obviously we get multiple urls like below, any url you try to access will be able to access web application, for suppose my web application is a votting app then I would like to create single url by using “Load Balancer Service” that looks like below .

NodePort Service url

“curl http://<node1IPaddress>:<Nodeport>”

“curl http://<node2IPaddress>:<Nodeport>”

“curl http://<node3IPaddress>:<Nodeport>”

“curl http://<node4IPaddress>:<Nodeport>”

Load Balancer Service url

“Curl http://example-vote.com”