**KUBERNETES**

**Docker**: A docker is a containerization platform.

**Kubernetes**: A Kubernetes is a container orchestration platform.

**Problems with Docker:**

* As the containers will be in a single host they might face some issues.
* If any container uses more CPU, then it will affect the other containers performance also.
* If a container is down then the we have to manually start the container again. **Auto Healing** is a concept that the containers automatically recreate even if they are terminated.
* During the peak load conditions, the applications get lots of requests so to accommodate this issue the no. of containers should be increased. This is called **Auto Scaling**. Docker cannot do these above things.
* Load balancing is also important.

Docker does not support enterprise level support which has requirements like the below

* Load balancing
* Auto Heal
* Firewall
* Auto scaling
* API Gateway

All these problems are solved by **Kubernetes**.

Kubernetes is a cluster means a group of nodes. Like a master node concept.

**Docker:**

Docker is a tool designed to make it easier to deploy and run the applications by using the containers.

Containers allows the developer to package an application with all the parts it needs such as libraries and other dependencies.

We use docker to create containers.

**Kubernetes:**

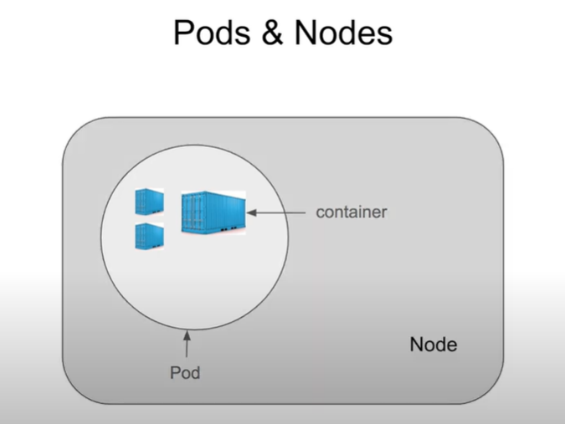
The Kubernetes is a container orchestration engine that manages the containerized applications like deploying, scheduling, load balancing, auto healing, availability, monitoring and scalability.

It is a container management tool.

We use Kubernetes to manage the containers.

**Pods and Nodes:**

**Pods**: A group of containers or a single container is called a pod in Kubernetes.

**Nodes**: A group of pods or a single pod can be deployed on a node. A node is virtual machine.

**Features of Kubernetes:**

1. **Automatic bin packing:**

Automatically places containers based on their resource requirements like CPU & Memory (RAM) while not sacrificing availability and saves resource.

If any containers needs large amount of CPU and storage then it will be kept in a server where the other containers utilizes less CPU and storage. Thus this helps in not affecting the other containers with similar requirements.

The requirements like CPU, memory ,etc of the containers are specified in a pod. These pods are analysed by the Kubernetes scheduler makes a decision to place the pod on a particular node.

1. **Service discovery and load balancing**

[Kubernetes Beginner Tutorial 2 | Features of Kubernetes - Part 1 - YouTube](https://www.youtube.com/watch?v=e6n5a0nB0PM)

1. **Storage orchestration**
2. **Self Healing**

**Kubernetes Architecture:**

Worker

Master

Kube-proxy

Kubelet

Container run time

Data Plane

Control Plane

* In kubernetes the containers are deployed as pods (a group of containers).
* The architecture contains of one Master and multiple workers. The Master is the Control plane consists of various components and the Worker consists of components like Container run time (containerD/cri-o), kubelet, kube-proxy, etc.

**Components of Data Plane:**

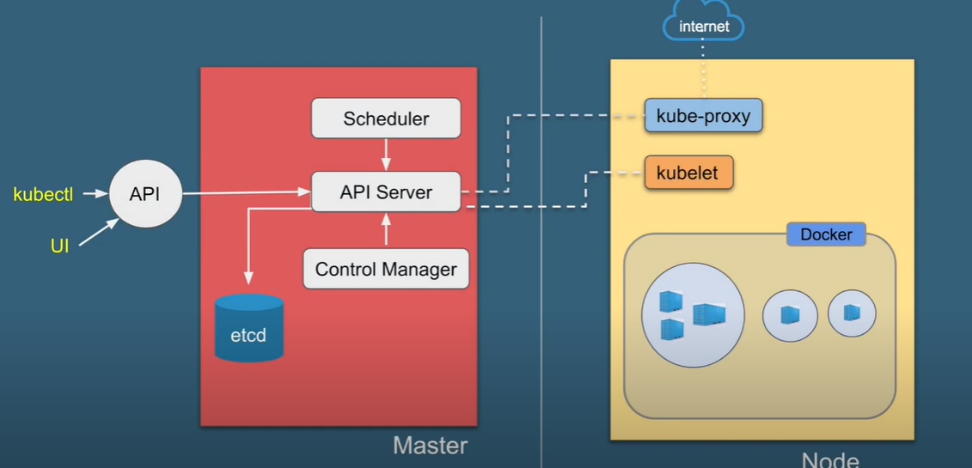
1. **Container run time:**

The containers need a run time environment for its operation just like java needs java run time. The container run time provided by kubernetes are Container D and Cri-O.

1. **Kubelet:**

The kubelet is responsible for maintaining the pod. If any issue is found then the kubelet will inform the auto healing in the control plane to rectify the issue. Creation of Pods and ensure the pods run always.

1. **Kube-Proxy:**

This provides networking for pods. The autoscaling is also done by kube-proxy. It is responsible for generating IP addresses , load balancing, etc. It uses IP tables in linux machine to perform the configurations.

**Components of Control Plane:**

1. **API Server:**

This is the core component of the Kubernetes. This takes all the incoming the requests like Identity related, security related, decision of pods deployment in nodes, etc.

This API server is exposed to the external world so it can be accessed.

1. **Scheduler:**

It is responsible for scheduling the pods and resources in the kubernets based on the node availability.

1. **Etcd:**

This stores the entire Kubernetes information in the form of objects. This is a backup of the kubernetes.

1. **Controller manager:**

The Kubernetes has autoscaling and it is done by a controller called Replica set. This replica set controller has to run continuously so to ensure this the controller manager will monitor and control this services.

1. **Cloud controller manager:**

An integration between the cloud platform and Kubernetes. This acts a bridge and perform the requests from the Kubernetes on the cloud platform.

If there is a request for the scaling of the resources then the Kubernetes will convert the request into APIs of the cloud platform like azure, aws or gcp and gets the services. CCM will make this happen.

**Kubernetes Production Distributions:**

Just like the linux distributions we have distributions for production in the Kubernetes also.

* Openshift
* Rancher
* Tanzo
* EKS
* AKS
* GKE
* DKE

**Terms:**

* **Cluster**: When you deploy a Kubernetes you get a cluster. A cluster is a set of machines called nodes. A cluster has at least one master node and worker node. Kubernetes supports not more than 5000 nodes in a cluster.
* **Nodes**: A node can be a physical machine or a virtual machine. Max 1,50,000 pods in a node. (max 100 pods per node)
* **Pod:** A pod can have one or more containers. Max 3,00,000 containers in a pod

**Kubectl:**

kubectl (often pronounced "cube control" or "kube control") is the command-line tool used to interact with Kubernetes clusters. It is an essential tool for managing Kubernetes, allowing users to control Kubernetes clusters by issuing commands that manage the deployment, scaling, and operation of containerized applications.

**Minikube:**

**Minikube** is a tool that allows you to run a local Kubernetes cluster on your personal computer. It's mainly used for development, testing, and learning Kubernetes without needing a full-fledged multi-node Kubernetes cluster or cloud provider setup. Minikube is lightweight and typically runs on a single node

**Commands:**

**kubectl cluster-info** -Get cluster information

**kubectl get all** -View all resources

**kubectl get pods** - Get a list of pods

**kubectl apply -f <filename>.yaml** -Apply a configuration from a file. After running this command, Kubernetes will create the deployment and launch the specified number of NGINX Pods.

**kubectl create deployment nginx --image=nginx** -Create a deployment

**minikube start** - A single-node Kubernetes cluster is created with a control plane (master node) and a worker node. The Kubernetes components, such as the API server, scheduler, and controller manager, are started.

**kubectl get pods** **-o wide**  -gives complete details of all the pods

**Basic Deployment in kubenetes:**

Kubernetes (K8s) is a container orchestration platform that automates the deployment, scaling, and management of containerized applications. A basic Kubernetes deployment involves creating a Deployment resource, which manages a set of identical Pods (running containers) and ensures that the desired number of Pods are running.

Here’s a breakdown of the steps to create a basic Kubernetes deployment:

Use Git BASH for CLI and open the docker desktop then only it will run.

**1. Set up a Kubernetes Cluster**

* You can use a local setup like Minikube, a managed Kubernetes service like Google Kubernetes Engine (GKE), Amazon EKS, or Azure AKS, or use tools like k3s for lightweight clusters.

Once your cluster is up and running, you can interact with it using kubectl, the Kubernetes command-line tool.

**Start Minikube**

Once Minikube is installed, you can start a local Kubernetes cluster:

minikube start

This command starts a single-node Kubernetes cluster on your local machine. You can check the status of the cluster with:

minikube status

**2. Define the Deployment**

A deployment is typically defined in a YAML file. Below is an example of a basic deployment YAML file:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

Explanation:

* apiVersion: Defines the version of the Kubernetes API to use.
* kind: Specifies the type of object being created, in this case, a Deployment.
* metadata: Includes the name and labels for the deployment.
* spec: Defines the desired state for the deployment.
  + replicas: Specifies the number of Pods that should be running.
  + selector: Matches the Pods to manage based on their labels.
  + template: Defines the Pod template for the containers.
    - containers: Specifies the containers that should run inside the Pods, including the image to use (nginx in this case) and ports to expose.

**3. Apply the Deployment**

Once you’ve defined the deployment YAML file (e.g., saved as nginx-deployment.yaml), you can create the deployment using kubectl:

kubectl apply -f nginx-deployment.yaml

This command tells Kubernetes to create a deployment as defined in the YAML file.

**4. Verify the Deployment**

To verify that the deployment is running correctly, you can use the following commands:

* Check the status of the deployment:

kubectl get deployments

* Check the Pods that were created:

kubectl get pods

* Describe the deployment to see detailed information:

kubectl describe deployment nginx-deployment

**5. Expose the Deployment**

To make the application accessible, you need to expose the deployment by creating a Service. You can create a service by running:

kubectl expose deployment nginx-deployment --type=LoadBalancer --port=80

This command exposes the deployment as a service and assigns a public IP (depending on the cloud provider or local settings).

**6. Access the Application**

Once the service is created, you can access your application by getting the external IP:

kubectl get services

For a local setup like Minikube, you might need to run minikube service nginx-deployment to access it.

**Difference between Container, Pod and Deployment:**

**Container:** A container is an executable package that has everything to run an application. We need to crate an image then we can build a container using the image and later we run the container.

Image

**Pod:**  Here the entire details of image, container, port to be exposed are written in a yaml file. A pod can have one or multiple containers. Both containers in a pod will have dependency.

A pod is a runtime specification of a container.

We can run the containers in pods and can create a network between two containers also.

YAML file

(pod.yaml)

**Deployment:** A deployment is the support system to the pods. The pods don’t have auto healing and auto scaling but the deployment has these features. The deployment will also deploy a pod but with features.

Here we can have auto healing and auto scaling in deployment only.

Creating a pod with a deployment resource is best option as it has good features.

The deployments come with replica sets (a controller). Like we can have multiple pods for load balancing, availability, etc. This entire is written in a yaml file.

The replica sets will auto heal the pods if any user deletes any of the pods. It will ensure that all the pods are running fine by deploying the deleted pod. If we set the replica count as 50 then the controller will make sure that 50 pods are running all the time. If any fails then it will automatically deploy the pod in place of the failed pod.

Replica

Sets

YAML file

(deployment.yaml)

Without replica sets we cannot do the deployments. But we can use the pods without replica sets.

Controllers are very important as they make the Kubernetes as enterprise level.

**Replica set:**

A replica set is a Kubernetes controller that implements the auto healing and auto scaling. This will ensure that the no. of pods required will be the same and even if any of the pod is down then the Replica set controller will create another pod and servers the requests.

In production the deploying of the deployments will create the replica sets and Replica sets will create the pods and ensure its functioning.

**Deploying a Deployment in Minikube: (Replicasets)(Auto Healing)**

* minikube start - This creates a cluster
* kubectl get all (will give the details of the cluster)
* minikube status (ensure all are running and configured

If not use command

minikube delete

minikube stop

minikube start

* create a folder
* vim deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

Here we have created three replicas of the application means three pods.

* kubectl apply -f deployment.yaml ( a deployment is created)
* kubectl get pods (gives 3 pods)
* kubectl get rs (rs= replica sets)
* kubectl get deployments (gives 3/3 deployments)
* kubectl delete pod nginx-deployment (this deletes the pod)
* kubectl get pods -w (this gives that watch of the pods like a debugging mode used in another terminal to eatch the termination and creation of the pod))
* kubectl get pods

the pods gets deleted and new pod is created.(Auto healing is performed)

* kubectl delete deployment (nginx-deployment)

this permanently deletes the entire deployment

**Creating service to deployment:**

Requirement of a service:

Deployment/ RS

Pod1=Pod4

Pod3

Pod2

Pod1

User 2

User 1

User 3

Service (payment.svc)

Pod1-192.168.1.1

Pod2-192.168.1.2

Pod3-192.168.1.3

Pod4-192.168.1.4

* Here these pods are deployed as a deployment and the Replica set controller is backing them up.
* Each pod is given with an IP address so that the users can access them.
* If we delete the Pod1 or the pod is down due to network issues then the Replica set will create a pod Pod4 in place of Pod1 with a new IP address.
* But this new IP address cannot be given to users as it is an automatic process and if huge number of pods are available then we cannot check the pods IPs and provide them to the users.
* As a solution for this a Service is created and the users will access the pods through this service. The service will create a domain name like **payments.default.svc**. The service will have an IP address.
* Instead of having all IPs of the pods we can access all the pods with only on IP of the service and this service will take care of the load balancing also
* Along with this the load balancing of the pods will be done. This is achieved by the Kube-proxy.

**Advantages of Service in kubernetes:**

* **Load balancing:**

It is done by sending the requests to all the pods equally so that all the pods will serves the user and reduces the load on a single pod.

* **Service Discovery:**

The service will also face problem in identifying the pods with IP address. So, for this the service will use the labels/selectors to identify the pods belonging to that particular service or deployment.

**Labels/Selectors:** These are included in the yaml file of the deployment so that the pods created by that particular deployment will have the same label or selector.

labels:

app: nginx

spec:

replicas: 3

Here the app:nginx is the label. Even if the pods goes and IP changes but the label stays the same.

* **Expose to world:**

This will allow the pods to be accessible to outside the Kubernetes cluster. In the Kubernetes we can access by ssh but others in some part of the world needs to access will be difficult.

For this the application should be exposed for external users.

**Types of services:**

Service type is determined in the yaml file.

**Cluster IP mode:**  Can be accessible only inside the cluster. Who ever have access to Kubernetes can access.

**Node port mode:** Can be accessible only inside the organization. (who ever has access to nodes can access the applications inside the node.

**Load Balancer mode:** External world accessible. Everyone can access through public IP address. This will work only through the cloud provider platforms not in minikube.

Using the **cloud controller manager** the load balancing service will work.

**Kube Proxy:**

A Kube proxy works by maintaining a set of network rules on each node in the cluster, which are updated dynamically as services are added or removed.

When a client sends a request to a service the request is intercepted by kube-proxy on the nodes where it was received. Kube-proxy then looks up the destination endpoint for the service and routes the request accordingly.

It ensures that services can communicate with each other.

**Kubelet:**

Kubelet is responsible for the maintaining the lifecycle of the pod. It will ensure that the pods are running without interruption.

The kubelet will continuously monitor the pods and if any pod goes down then the kubelet will inform the replica set controller to deploy a new pod as a replacement thus maintaining the zero-down time of the applications.

* A pod by default will have cluster IP mode of service.

**Creating a service for the deployment with Node port mode and Load balancer mode :**

1. Start minikube and create a cluster and check the status

* minikube start
* minikube status
* kubectl get all

if any error

* minikube delete
* minikube stop
* minikube start

1. Creating a docker file and application code file

* mkdir Kubernetes
* cd Kubernetes
* touch index.html
* vim index.html

<html>

<center>

Who is the GOAT?

</center>

<body>

<p> I am the greatest of all time </p>

</body>

</html>

* touch Dockerfile
* vim Dockerfile

FROM nginx

COPY index.html /usr/share/nginx/html

EXPOSE 80

1. Creating a docker image and connect the docker with minikube environment

* eval $(minikube docker-env)
* docker build -t firstapp .

1. Creating a file named deployment.yaml

apiVersion: apps

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: firstapp

imagePullPolicy: IfNotPresent

ports:

- containerPort: 80

1. Checking the deployment, pods, cluster

* kubectl apply -f deployment.yaml
* kubectl get deploy
* kubectl get pods
* kubectl get svc
* kubectl get rs
* kubectl get pods -o wide

1. Checking with default cluster ip mode

Copy the ip address of any one of the pods

* minikube ssh
* curl <http://128.58.3.9:80>
* logout

1. Creating the service.yaml file

**apiVersion**: v1

**kind**: Service

**metadata**:

**name**: my-service

**spec**:

**type**: NodePort

**selector**:

**app.kubernetes.io/name**: nginx

**ports**:

- **port**: 80

**targetPort**: 80

**nodePort**: 30007

kubectl apply -f service.yaml

kubectl get svc

1. Checking the service from nodes (our laptop).

curl http://180.16.14.1:30007

Post creating the service we can edit it with the below command

kubectl edit svc servicename

and change Nodeport type to LoadBalance type. This will allows you to access your application from external world.

It is possible only through communication with the cloud platform and it is done by the **Cloud controller manager**.

* If any changes in the selector or the label name then the service detach it self to the pods means we cannot access the pods with the service we have created.
* The traffic is controlled by the **Kube proxy** and divides the traffic among the pods equally.
* If any pod is down or deleted then the Kubelet will create a new pod to replace the terminated pod. As the newly created pod will have same selectors/labels so this kube proxy will detect it and adds it to the correct service of the application.

**Kubernetes Ingress:**

In Kubernetes, **Ingress** is an API object that manages external access to services within a cluster, typically HTTP and HTTPS traffic. It allows you to define rules for routing requests from outside the cluster to services inside it. An Ingress can provide load balancing, SSL termination, and name-based virtual hosting.

**Key Components of Kubernetes Ingress:**

1. **Ingress Controller**: A component that processes Ingress resources and configures a load balancer according to their rules. It’s responsible for fulfilling the Ingress resource by provisioning resources like load balancers and routes. Popular Ingress controllers include:
   * NGINX Ingress Controller
   * HAProxy Ingress Controller
   * Traefik
   * AWS ALB Ingress Controller (specific to AWS)
2. **Ingress Resource**: A configuration (YAML) that defines the rules and routing policies. For example, you can use Ingress resources to route traffic based on the hostname, URI paths, or HTTP methods.

We need to install **Ingress Controller** and create and run **Ingress Resource.**

**Benefits of Using Ingress**

* Consolidates multiple services under one external IP.
* Provides SSL termination and HTTP to HTTPS redirection.
* Supports complex routing rules, such as path- or host-based routing

**Steps to Set Up an Ingress:**

1. Deploy an Ingress Controller in your cluster (if not already deployed).
2. Define the Ingress resource with rules to manage routing.
3. Apply the Ingress resource and check if the external traffic is routed as expected.

**Basic Ingress File:**

* Vim ingress.yaml

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: example-ingress

spec:

rules:

- host: example.com

http:

paths:

- pathType: Prefix

path: “/bar”

backend:

service:

name: example-service

port:

number: 80

* kubectl apply -f ingress.yaml

Here the ingress is created.

Now we have to configure the ingress control of your choice as it depends upon the load balancer vendor you want to use.

We take nginx as it is light weight.

* minikube addons enable ingress

After doing this we have installed the nginx ingress controller. This will identify the ingress we have created earlier and sync with that, it will give an IP address to the ingress we have created earlier.

Till the configuration of the ingress controller the ingress will not get the IP address.