Kubernetes Overview

Why Kubernetes

- Consider a service running on Docker containers across multiple machines in your data center. If any of these containers or their host machines fail, your service becomes inaccessible, leading to a production outage.
- Managing massive applications with multiple containers poses challenges, such as networking, exposing APIs, and allowing user access. While containers and virtual machines can address these issues, achieving fault tolerance and high availability requires orchestration.
- **Orchestration**: This concept enables seamless collaboration between multiple containers or microservices. Docker Swarm and Kubernetes are popular orchestration tools that solve this problem.
- For large-scale applications where scalability and high availability are crucial, orchestration is often the solution, making Kubernetes a necessary tool.

Pod

- A Pod is the smallest unit in Kubernetes, similar to how a container is the smallest unit in Docker.
- In Kubernetes, containers are created inside a Pod (one or many).
- A Pod can contain shared volumes, multiple containers, and a shared IP address.
- Typically, one container per Pod is used in the industry unless there is a specific use case for multiple containers.
- Each Pod must reside on a single server.
- Pods can be moved from one node to other node in the cluster.
- Each container in a pod share namespaces in that pod like the IP addresses or the shared volumes

Kubernetes Cluster

- A Kubernetes cluster is a group of machines (physical or virtual) that run containerized applications.
- Each machine/server in the cluster is called a node, and there can be multiple nodes in a cluster.
- Each node has one or more Pods, and each Pod usually contains one container. Pod is the smallest deployable unit in Kubernetes.
- A Kubernetes cluster has a Master node and worker nodes. The Master node manages the worker nodes.
- **Master Node**: The Master node's job is to manage load across worker nodes. All Pods and containers that run your application are deployed on the worker nodes.

- The Master node is the control plane and does not run client applications. It runs essential components to keep the cluster operational and to manage it.

Components in Control Plane Node

- **API Server**: The central component of the Kubernetes control plane that exposes the Kubernetes API. Using the API server, you can manage entire cluster and this can be done using kubectl or kube control.
- **Scheduler**: A service on the Master node that manages load distribution between the nodes in the cluster. The scheduler receives the request from the API server and it decides which node to deploy the pod on.
- **Kube Controller Manager**: Manages everything in the cluster in terms of what should happen on each node.
- **Cloud Controller Manager**: Works with the cloud service provider to scale applications and perform other automated tasks.
- **etcd**: A distributed key-value store that stores the state of the cluster. It is used to store the logs and errors in the cluster. It stores data in a key-value pair format. Stores data in JSON. Any change made to the cluster is stored in etcd. It is a distributed database. It is used to store the state of the cluster. Only the API server can write to etcd. All other components can only read from etcd.
- **Kube CTL**: This is a command line tool that you can use to manage the cluster using command line. The way kubectl works is that the REST API communicates with the API server on the master node. The communication happens over HTTPS.

Components in Worker Node

- **Kubelet**: An essential component in a Kubernetes cluster that communicates with the API server on the Master node to receive commands to launch or destroy containers. It is the agent that runs on each node in the cluster. It is responsible for communication between the worker node and the master node.
- **Kube-Proxy**: Present on each node and used for network communications between all nodes.

Workflow

- The way the workflow in Kubernetes works is that when an administrator wants to deploy a pod on the cluster, they send a request to the API server on the Master node. The request to the API server is sent using the KubeCTL command line tool. The API server authenticates and validates the request made by the administrator.
- The API server then sends the request to the scheduler. The scheduler then decides which node to deploy the pod on and communicates this information back to the API Server.
- The API server then sends the information to the Kubelet on the node that the scheduler decided to deploy the pod on. Kubelet then launches the container on the node and communicates the information back to the API server.
- The API Server then sends this information to the etcd on the Master node and etcd stores this information. The API Server also communicates the same to the administrator stating that the pod has been successfully deployed.

Learning Setup

- **Install MiniKube**: MiniKube is a local kubernetes cluster which lets you setup a cluster locally and have your manager node and worker node on a single machine
- **Install Kind**: Just like minikube, kind is a tool for running local kubernetes clusters using Docker container. It is a more powerful tool than minikube and can be used to run more complex clusters. To install it on MacOS you can use Homebrew and run the following command:

  `brew install kind`
- **Install kubectl**: kubectl is the command line tool that you can use to manage the cluster.
- **Create a Clutser**: Once you have installed kind and KubeCTL, run the following command to create a cluster

  `kind create cluster --name test`
- The above command will create a cluster with the name test. This will be a single node cluster running the control plane and other essential components and also lets you run pods. You can then use the following command to get the cluster information.

  `kubectl cluster-info`
- **Multi-Node Cluster**: You can also deploy a multi node cluster using kind. To do this, you need to create a configuration file that defines the nodes in the cluster and then create the cluster.
- **Config File**: Save the below configuration in a file called anyname.yaml.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
```

- **Create Multi-Node Cluster**: Once you have the configuration file, you can create the cluster using the following command

  `kind create cluster --name anyname --config anyname.yaml`
- **Verify Cluster**: Once the cluster is created, you can verify it using the following command

  `kubectl cluster-info --cluster=anyname`
- Since you now have two clusters - a single node and a multi-node cluster, you need to specify a context for kubectl to use and hence you specify it using ***--cluster=clustername*** to get info about the cluster.
- **Contexts**: To see the contexts currently available, you can use the following command. The star symbol indicates the currently set context.

  `kubectl config get-contexts`

- **Set Context**: To set a context to a specifc context, you can use the following command

  ```
  kubectl config use-context anyname
  ```

- Now that you have a cluster set up, it is time to create some pods and look at multiple ways to create them

Creating Pods: Imperative & Declarative Way

- **Imperative Way**: The imperative way of creating pods is by using the ***kubectl run*** command. This command is used to run a pod using a simple command and it is the most basic way of creating pods. For example, to create a pod that runs a container with the name nginx, you can use the following:

  ```
  kubectl run nginx --image=nginx:latest --port=80
  ```

- The above command creates an nginx pod with the latest version of the nginx image and exposes port 80.

- **Declarative Way**: This is done using a yaml configuration file. The yaml file defines the pod and its configuration. For example, to create a pod that runs nginx we can specify multiple metadata information in the yaml file. Navigate to nginx.yaml file