Unit-4

Docker & Kubernates

Introduction to Docker

Docker is a open source tool or platform that allows developers to build, package, and deploy application in a lightweight, portable container. These docker containers bundle and application with all its dependencies, libraries and configuration files, ensuring it runs consistently across different environment.

Features of Docker

- **Containerization:** Divide application into small and portable container
- **Portability**: Containers run anywhere
- **Light-weight:** All containers are lightweight.
- **Scalability:** We can change the size of container.
- **Fast Deployment:** Faster to start and deploy than VMs.

Disadvantage of Docker

1. Complexity at Scale

- Managing containers at scale (hundreds or thousands) can get complicated quickly.
- Requires orchestration tools like Kubernetes, which have their own steep learning curves.

2. Security Concerns

- Containers share the host OS kernel, which can lead to security vulnerabilities if not properly isolated or configured.

3. Tooling Overhead

- Docker might require integration with multiple other tools (CI/CD, logging, monitoring, etc.), adding to the overall DevOps stack complexity.

4. Not Ideal for GUI app

Docker Command

1. Run Command: run a container from a image

```
$ docker run <image_name>
$ docker run --name <container_name>   <image_name>
```

2. Pull Command: To pull a image from docker regitory

```
$ docker pull <image_name>
```

3. Docker PS: Listout the all Docker images

```
$ docker ps
```

4. Docker Stop: To stop a running container

```
$ docker stop <container_ID>
```

5. Docker start: To start or Run a container

$ docker start <container_ID>

6. Docker rm: To delete a container

$ docker rm <container_ID>

7. Docker rmi: To delete a docker image

$ docker rmi <image ID/ image name>

Docker Installation and Setup

Steps of Docker installation in Windows Operating System and Mac Operation System.

Step-1: Open any browser and Visit Docker Official Website.

Step-2: Download windows installer of Windows OS or Drag Docker into your Applications folder for mac OS

Step-3: Run the installer for windows and follow the instruction prompt. But in Mac OS just launch the docker

Step-4: After installation run from Start menu or Whale ICON

Steps of Docker installation Linux

Open linux Terminal and Run the given command

- sudo apt update
- sudo apt install docker.io -y
- sudo systemctl start docker
- sudo systemctl enable docker

Working with Docker Image and Container

Docker Image: A Docker image is a lightweight, standalone, and executable package that includes everything needed to run a piece of software,

Including:

- The code or application itself
- Runtime (e.g., Node.js, Python)
- System tools and libraries
- Dependencies
- Default environment variables
- Any configurations needed

A Docker image is built in layers. Each instruction in a Dockerfile (like RUN, COPY, FROM) creates a new layer. Layers are cached and reused to make building efficient.

- FROM node:22.14.0 #show a node version for base image
- WORKDIR /app #it connect to a js directory file
- COPY . /app #Copy the js file for image

- RUN npm install #Install dependencies
- EXPOSE 8080 #Represent port Number
- CMD ["node" "index.js"] #command for run the App

Docker Container: A Docker container is a running instance of a Docker image. It is a light-weight, Isolated, portable instance.

- Image = The blueprint (like a recipe)
- Container = The dish made from the recipe (alive and running!)

To Run a Docker image into a Container

docker run -d --name my-container -p 3000:3000 my-node-app

- Docker run: Command for Run a image to a container
- -d: Run in detached mode
- --name: container name
- -p: Port number where you want to run

Dockerfile and image Creation

Dockerfile: A Dockerfile is a text file that contains a set of instructions used to build a Docker image. It uses a Domain Specific Language (DSL) to define the steps required to create an image, which can then be used to run containers. Dockerfiles are essential for automating the process of creating Docker images, ensuring consistency and reproducibility across different environments

Structure of Docker file

```
my-app/
├── Dockerfile
├── package.json
├── package-lock.json
└── index.js
```

Set of Instruction of Docker file

- FROM node:18 # Use the official Node.js base image
- WORKDIR /app # Set the working directory inside the container
- COPY package*.json ./ # Copy package.json and install dependencies
- RUN npm install
- COPY . . # Copy the rest of your app's code
- EXPOSE 3000 # Expose the port No.
- CMD ["node", "index.js"] # Command to run your app

➢ For image Creation in Docker from a docker file, we have a command only we have to navigate the file directory and run the given command for image creation

docker build -t my-node-app .

➢ And After that you can check your image is created or not using given command

docker images

Docker Compose and Docker Swarm

Docker Compose: Docker Compose is a tool that allows developers to define and run multi-container Docker applications using single YAML file. It is most commonly used in development, testing, and staging environments. Its primary purpose is to simplify the orchestration of containers that work together within a single application.

Docker Compose YAML STRucture

```
version: '3'
services:
  web:
    image: node:18
    volumes:
      - .:/app
    working_dir: /app
    ports:
      - "3000:3000"
    command: node index.js
  db:
    image: mongo
    ports:
      - "27017:27017"
```

How Docker Compose Works

Imagine an application that has a frontend, backend, and database. Using Docker Compose, you can define all three components and their relationships in one YAML file. E.g. services, networks, and volumes in "docker-compose.yml"

After that we can directly run everything using a single command

"docker-compose up"

Features of Docker Compose

- **Declarative Configuration:** Uses a YAML file (commonly docker-compose.yml) to configure application services. This file specifies how to build the container images, configure volumes, networks, and set environment variables.
- **Multi-Container Setup**: Define and manage all the containers needed for an application in one file.
- **Effortless Networking**: Provides an isolated network where containers can communicate with each other by default, using their service names as DNS.
- **Portability:** You can easily share the configuration file, allowing the application to run consistently across different environments.
- **Command-Line Interface:** Simple commands like docker-compose up to start services or docker-compose down to stop them.

Docker Swarm

Docker Swarm is a container orchestration tool, but unlike Docker Compose, it is designed for managing clusters of Docker nodes in production environments. It provides fault-tolerant and distributed deployments for services at scale.

Docker Swarm Command for init and Deploy.

- Initialize a Swarm: docker swarm init
- Deploy a Stack: docker stack deploy -c <compose-file> <stack-name>

Features of Docker Swarm

- **Cluster Management:** Converts multiple Docker nodes into a single logical cluster. Nodes can be designated as Manager Nodes (orchestrating the cluster) or Worker Nodes (executing tasks).
- **Service Scaling:** Easily scale services up or down by specifying the number of replicas.
- **Load Balancing:** Automatically balances traffic among replicas of a service.
- **High Availability:** Ensures services remain active, even if a node goes offline.
- **Secure by Default:** Uses TLS encryption for communication between nodes in the cluster.
- **Declarative Configuration:** Define desired service states (e.g., how many replicas) in YAML files or directly via the Swarm CLI.

Differences

Feature	Docker Compose	Docker Swarm
Purpose	Local development and testing	Production-grade container orchestration
Scale	Single-node setups	Multi-node clusters
Networking	Local network	Distributed overlay network
CLI Simplicity	Simple commands for basic needs	Comprehensive commands for orchestration
Fault Tolerance	Not inherently fault-tolerant	Built-in high availability

Orchestration with Kubernates

Introduction to Kubernates

Kubernetes is a Open Source Automated Orchestration tool that is used to automate the deployment, scaling, and management of containerized applications. Originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes enables you to run containers at scale across multiple machines, ensuring your applications are resilient, efficient, and highly available.

Features of Kubernetes

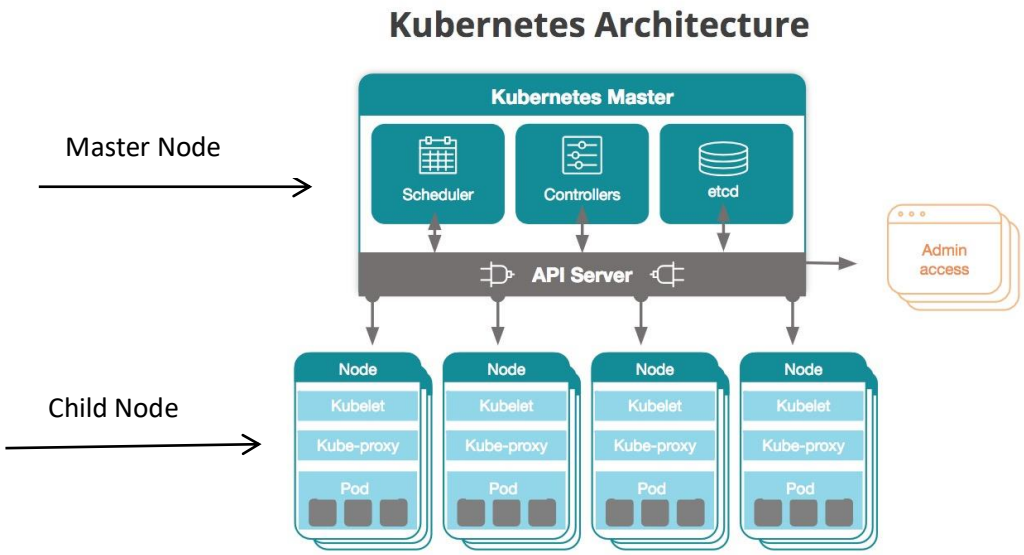
- **Container Orchestration:** Kubernetes schedules and manages containers across a cluster of machines. It ensures the right containers run on the right nodes and handles their lifecycle.

- **Service Discovery and Load Balancing:** It provides networking capabilities to automatically expose services. Kubernetes can load-balance traffic and route requests to healthy containers.
- **Scaling:** Automatically scales your application up or down based on demand. This can be done manually or via auto-scaling rules.
- **Self-Healing:** If a container fails, Kubernetes restarts it. If a node crashes, Kubernetes reschedules containers to other healthy nodes.
- **Declarative Configuration:** Kubernetes uses YAML or JSON configuration files to define the desired state of your system, such as how many replicas of an application should run.
- **Storage Orchestration:** Dynamically mount and manage storage for containers, including support for persistent storage.
- **Rollouts and Rollbacks**: Manage updates to your application with minimal downtime. Roll back to a previous version if something goes wrong.

Why Organization Use Kubernates

- Kubernetes is ideal for large-scale containerized applications that require high availability, fault tolerance, and scalability.
- It works seamlessly with cloud providers (like AWS, Azure, GCP) and can also run in on-premises environments.
- Many organizations rely on Kubernetes for modern microservices-based applications.

Kubernates Architecture and Components



Components

Master Node: The brain of the cluster, responsible for managing the desired state of the system.

Key components:

- **API Server:** Exposes the Kubernetes API.
- **Controller Manager:** Ensures the desired state matches the actual state.
- **Scheduler:** Allocates resources by assigning containers to nodes.
- **etcd:** A distributed key-value store used for storing configuration and state data.

Worker Nodes: Run the containerized applications and are managed by the master.

Key components:

- **Kubelet:** Ensures containers are running.
- **Kube:** proxy: Manages networking.
- **Pods:** The smallest deployable units, each containing one or more containers.

Setting up a Kubernates Cluster

Cluster: A Kubernetes Cluster is a collection of machines or node that work together to run and manage containerized applications. It consists of two main types of nodes: The Control or Master Node and Worker Nodes. Together, they provide a scalable and fault-tolerant environment for deploying, managing, and scaling containers.

- Setup a kubernates Cluster can vary on different types of Machines like. Local Machine, Cloud services and Virtual Machines.
- So In simple term, we can build a Kubernates cluster based on our need or Machines. And we have to setup an Environment for each type of machine. We have three common types of Setup
  - Cloud Based Setup.
  - Minikube based Setup. (For local system)
  - Using Kubeadm based Setup. (Bare Metal/Best for production)

1. Cloud Based: Cloud Based Cluster are fully managed or it can be created by Cloud web UI or CLI.
We have many types of Cloud like.

- AWS → AWS EKS (Elastic Kubernates Services)
- Azure → Azure AKS (Azure Kubernates Services)
- Google → GKE (Google Kubernates Engine)

Setup on GKE:

For Create a Cluster with Zone

- Step-1: gcloud container clusters create my-cluster --zone us-central1-a
  After Creating a Cluster we have to create a Pod
- Step-2: Kubectl run --image tomcat webserver
  After creating cluster and Pods we can check our cluster is working or not using given command
- Step-3:
  - kubectl get nodes
  - kubectl get pods -A
  - kubectl create deployment nginx --image=nginx
  - kubectl expose deployment nginx --port=80 --type=NodePort
  - kubectl get svc

Note: In a pods we have an option to define our steps or machine configuration using .YAML file like. Container type, Container name and Target port number

Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: jenkins-pod
spec:
  containers:
  - name: myjenkins
    image: jenkins/jenkins
    ports:
    - containerPort: 8080
      hostPort: 8080
```

Setup on Minikube:

Minikube runs a single-node Kubernetes cluster on your local machine. For execute a Minikube on local system we need Docker Application, Kubectl CLI, Minikube application.

Steps:

- Open Minikube and start the server
- Open Kubectl CLI and Run Given Command
- kubectl create deployment hello-minikube --image=k8s.gcr.io/echoserver:1.4
- kubectl expose deployment hello-minikube --type=NodePort --port=8080
- Now Open any browser and access your cluster using provided port number.

Managing Pods, Deployment and Services

Managing Pods in Kubernetes

Pods are the smallest deployable units in Kubernetes. They encapsulate one or more containers, along with shared storage and networking resources**.**

Pod Management task in kubernates.

1. Creating a Pod:

In Kubernates, we can create a pod with or without task, if we want to create a pod with some task kubernates provide yml file to define one or more than one task in single file.

YML file Structure

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
```

Apply the configuration in yml file → kubectl apply -f my-pod.yaml

2. Viewing Running Pods

- kubectl get pods

3. Deleting a Pod

- kubectl delete pod <pod-name>

Managing Deployments in Kubernetes

Deployments ensure that an application is running the correct number of Pods and allow for automated updates, scaling, and rollbacks.

1. Creating a Deployment: For creating a deployment in kubernates we have a yml script to assigned a deployment task to kubernates server after that it provide us a automation in deployment.

YML File

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: nginx-container
        image: nginx:latest
        ports:
        - containerPort: 80
```

Apply the Deployment → kubectl apply -f my-deployment.yaml

2. Scaling Deployments : We can Scale up or down our deployement dynamically using given command

- kubectl scale deployment my-deployment --replicas=5

3. Rolling Updates: We can perform rolling update task of our deployment using given command

- kubectl set image deployment/my-deployment nginx-container=nginx:1.21

4. Deleting a Deployment: We can delete our deployment using given command

- kubectl delete deployment my-deployment

Managing Services in Kubernetes

In Kubernates, Services expose Pods internally within the cluster or externally to users. They provide network accessibility and load balancing for applications.

Types of Services

- ClusterIP (default): Exposes a Service only inside the cluster.
- NodePort: Exposes a Service on each Node's IP, accessible externally.
- LoadBalancer: Uses a cloud provider's load balancer to distribute traffic.
- ExternalName: Maps a Service to an external domain name.

1. For Creating a Service: For create a service in kubernates we have a yml structure.
Structure:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

2. Apply the Service

- kubectl apply -f my-service.yaml

3. For view a Services

- kubectl get services

4. Deleting a Service

- kubectl delete service my-service