Docker - LAB

#Create a Ubuntu VM
#Install Docker on VM
#Start and enable Docker

- sudo apt -y install docker
- sudo apt install docker.io

#Start and enable docker

- sudo systemctl start docker
- sudo systemctl enable docker

#Once enabled, provide permissions for docker-sock

- sudo chmod 666 /var/run/docker.sock

#Working with images

- docker images #will display available images
- docker pull <image-name> #will download image
- docker run <image-name> #will download image and run the image

#Listing Container

- docker ps #will display running cont
- docker ps -a #will display all cont

#When we start a container, it runs and it stops right-away
#Container's are ephemeral in nature.
#That means a container will not run without a purpose.

#Container Run-Modes

#Container can be executed in two Modes

- Interactive Mode (-it)
- Detached Mode (-dt)

Interactive Mode

- docker run -it ubuntu

#This will run the container in Front-End, we will be connected with the cont
#If we exit the cont, the cont stops running.

Detached Mode

- docker run -dt ubuntu

#This will run the cont, in the detached-backend mode.

#Communication With Container

- docker exec -it <cont-id> /bin/bash

#This will place us in the cont

- docker exec -it <cont-id> command

#This will execute the command within the cont

#Find Containers details

- docker inspect <cont-id> / <cont-name>

#Will show the complete details of the cont

#Control Container Status - Stop / Start / Delete Cont

- docker stop <cont-id> / <cont-name>
- docker start <cont-id> / <cont-name>
- docker rm <cont-id> / <cont-name> #will remove stopped cont
- docker rm -f <cont-id> / <cont-name> #will remove any cont

#Container - Naming

- docker run -dt - -name <cust-name> <image-name>

#This will create a cont, with a custom name

#Container-Porting

- Port forwarding : -P #will define a random port
- Port Assigning : -p #will assign a custom port

Port-Forward

- docker run -dt –name web -P <image>

Port-Assign

- docker run -dt --name web3 -p 8080:80 httpd

#Docker Networking

- Linux Container Networking
- Bridge, Host, None and overlay networks
- List, create, inspect and remove new networks.
- Launch container in desired network.

Default Networks

#Network available on docker host upon creation

- Bridge
- Host
- None

Command - docker network ls
Run - ip add show or ifconfig to see the network config

Bridge - Network

- In Docker, a bridge network is a default networking setup that allows containers running on the same Docker host to communicate with each other and access the internet. It essentially creates a virtual network within the host machine, enabling containers to connect and exchange data.
- Docker Network Modes Explained: Bridge, Host, and Overlay ...

Key aspects of Docker bridge networks:

Default Network:

- When you create a container without explicitly specifying a network, it's automatically attached to the default bridge network (named "bridge").

Isolation:

- Bridge networks isolate containers from the host machine's network and from other containers that aren't on the same bridge.

Communication:

- Containers within the same bridge network can communicate with each other using their IP addresses assigned by Docker.

Access to Host and Internet:

- Containers on a bridge network can access the host machine's network (if the host has network access) and the internet.

Customization:

- You can create custom bridge networks using the docker network create command.

Service Discovery:

- Docker provides service discovery for containers on the same bridge network, meaning they can find each other by name.

Single-Host Scope:

- Bridge networks are local to a single Docker host. If you need to connect containers on different hosts, you'll need to use an overlay network, as explained in the Docker documentation

Bridge-Network - LAB

- docker network inspect bridge #to get info about this network

- Run two containers, inspect the bridge. We can see our cont are listed under bridge network
- Internal addresses are assigned automatically, through DHCP, in private range
- To expose these cont/network we use porting option
- Port-forwarding Or Port-Assigning

Host-Network

- In Docker, the "host" network mode allows a container to directly share the network namespace with the host machine. This means the container uses the same network stack as the host, bypassing the usual Docker network isolation. Containers using host networking don't receive their own IP addresses and can access the host's network interfaces directly.

Shared Network Namespace:

- The container uses the host's network stack, including IP addresses, routing tables, and network interfaces.

No Port Forwarding:

- When a container using host networking listens on a port, that port is directly accessible on the host's IP address, without the need for port mapping.

Direct Access to Host Resources:

- The container can access host resources and communicate directly with other containers on the host, as they share the same network.

Higher Performance:

- Host networking can offer better network performance as there's no overhead from Docker's network bridge or other intermediary layers.

Reduced Isolation:

- Since the container shares the host's network namespace, it's not as isolated as with other network modes like "bridge".

Example:

- If you run a container using host networking and it binds to port 80, the application inside the container will be directly accessible on port 80 of the host machine's IP address

Host-Network-LAB

- Containers behave just as any other process running in the docker host
- The host network adds a container on the host's network stack
- There is no isolation between the host machine and the container
- Does not perform any operations on incoming traffic

- docker network inspect host
- docker run -dt –name web1 --net=host tomcat
- docker run -dt --net=host --name web1 httpd

#will expose the cont on the default port on the ec2

None - Network

- In Docker, the "none" network mode completely isolates a container from any network, meaning it cannot communicate with other containers or the outside world. This provides maximum security by sandboxing applications that don't need network access. The container only has a loopback device (127.0.0.1), enabling internal process communication.

- Complete Isolation: Containers are isolated from the host network and other containers.
- No Network Interface: No network interface (like eth0) is created within the container, only the loopback.
- Loopback Only: The container can only communicate with itself via the loopback address (127.0.0.1).

- Security: Useful for applications that don't need network access or when you need to isolate sensitive processes.
- No External Connectivity: Containers cannot reach the internet or other containers.

When to use "none":

- Security: When you want to isolate containers for maximum security, especially when handling sensitive data or processes.
- Batch Jobs: For running batch jobs that don't require network communication.
- Data Processing Pipelines: For processing data without external dependencies

LAB
#We will launch a container under None-Network
#Then we will switch the container from one network to another

- docker run -dt --name tomcat --net=none -p 8080 tomcat

#This will run the cont in none network and will not generate any ip addr

Disconnect and Add Network

- docker network disconnect none <cont-id> or <cont-name>

#This will disconnect the cont from none network

- docker network connect bridge <cont-id> or <cont-name>

#This will connect the cont to a new-network

Overlay Network

- In Docker, an overlay network is a virtual network layer that enables containers running on different Docker daemons to communicate with each other. It's used primarily in Docker Swarm to create a distributed network across multiple hosts, facilitating service discovery and communication between containers in a cluster. Overlay networks create a flat virtual network, meaning containers can communicate as if they are on the same network, even if they are physically located on different hosts.

Creates a distributed network:

- Overlay networks allow containers on different Docker daemons to communicate as if they were on the same network, simplifying application architecture.

Enables service discovery:

- Docker Swarm uses overlay networks for service discovery, allowing services to find each other within the cluster.

Secure communication:

- When encryption is enabled, overlay networks provide secure communication between containers.

Built-in with Swarm:

- Overlay networks are a key component of Docker Swarm, a container orchestration tool.
- Supports multiple networks:
- Services or containers can be connected to multiple networks simultaneously, allowing for flexible communication patterns.

How it works:

Overlay network driver:

- Docker uses the overlay network driver to create and manage the virtual network.

VXLAN technology:

- Overlay networks often utilize VXLAN (Virtual Extensible LAN) to encapsulate network traffic, enabling it to travel across the underlying network infrastructure.

Routing:

- Docker transparently handles routing of network traffic to the correct Docker daemon and container.

External K/V store:

- While older versions of Docker required an external K/V store (like Etcd) to manage overlay network state, Docker Engine 1.12 and later versions integrated this state management into the Docker engine.

OVER_LAY - LAB

- docker network create –driver=bridge new_bridge1

#Create a new bridge net-work

- docker network create --driver=bridge --subnet=172.99.11.0/24 new_bridge2

#Create a new bridge-network with custom subnet values

Moving Across Net-Works

- docker run -dt --name test --net=bridge -p 80 nginx

#Launch a cont in default network

#Disconnect the cont from default network

- docker network disconnect bridge 1a6c6f1f1e72

#Connect the cont to a new-network

- docker network connect new_bridge1 1a6c6f1f1e72

Removing - Networks

- docker network rm bridge

bridge is a pre-defined network and cannot be removed.

- docker network rm new_bridge1

#This will delete the network

- docker network prune

#This will delete all the unused networks

Multiple-Networks In containers

- docker network create –driver=bridge new_bridge1
- docker run -dt --name tc --network=bridge -p 80 nginx

#Now, we can add additional network to the same container

- docker network connect c2ee988ad5c9 89dba6e6cd27

Docker Volumes

- Docker storage concerns how data is managed and persisted within Docker containers. Because containers are ephemeral by design, meaning their data is lost when they are removed, Docker provides mechanisms to manage persistent data. Docker volumes are the preferred way to persist data generated by and used by Docker containers. They are managed by Docker and are stored in a dedicated directory on the host, typically /var/lib/docker/volumes on Linux systems.

Volumes:

- Docker-managed storage that persists data even after containers are removed. Volumes are ideal for persistent storage and data sharing between containers.

Bind mounts:

- Mounts a file or directory from the host system into a container. Changes made in the container are reflected on the host and vice versa.

tmpfs mounts:

- Stores data in the host's memory, offering faster read/write speeds but data is lost when the container stops.

Docker Volumes

Creation and Management:

- Volumes can be created using docker volume create and managed using commands like docker volume ls, docker volume rm, and docker volume prune.

Persistence:

- Data in volumes persists beyond the lifecycle of containers, making it suitable for stateful applications.

Sharing:

- Multiple containers can mount the same volume, facilitating data sharing between them.

Storage Location:

- Volumes are stored on the host filesystem within Docker's data directory.
- Use Cases:
- Volumes are suitable for databases, application data, and scenarios requiring data persistence and sharing.

Persistent-Storage

- Persistent storage refers to data storage that retains information even when the device or system is powered off or interrupted. It's also known as non-volatile storage, meaning data is not lost due to a power outage or other disruptions. This is in contrast to volatile storage, like RAM, which loses data when power is removed.

Volume-Types

- Unnamed volumes
- Named Volumes
- Host Volumes

#Unnamed-Volumes

- docker run -dt --name demo-vol -p 8090:80 -v /app-data httpd

#We cannot identity which vol belongs to which container

#Named-Volumes

- docker run -dt --name demo-vol -p 8090:80 -v vol-name:/app-data httpd

#These volume are managed by docker, we do not want this to happen
#We need to be able to manage our own volumes.

#Host-Volumes

- Host volumes are directly managed by the admin, where the data is stored on the Host-machine and it is mounted to the folder on the docker-container
- docker run -dt –name host-vol -v /home/ubuntu/app-data:/app-data -P httpd

#This will attach the volume on my host-machine to the volume on the container, whenever there is any change on the container volume, the change will be reflected on the host-machine. If the container is deleted, then the volume remains on host-machine and we can mount this volume to a new-container.

Docker Images - Dockerfile

- A Docker image is a read-only, template-like file that contains all the instructions needed to create a Docker container. It essentially acts as a blueprint for a container, including the application code, libraries, tools, dependencies, and other files required for the application to run.

Template:

- Docker images are templates, meaning they're not meant to be run directly, but rather used to create instances of containers.

Instructions:

- They contain a set of instructions, typically defined in a Dockerfile, that guide Docker on how to build the image.

Shareable and Portable:

- Docker images are designed to be shared and deployed across different environments, making it easy to run applications consistently.

Layered Structure:

- Images are built up in layers, which helps to speed up the building process, increase reusability, and reduce disk usage.

Starting Point:

- Docker images serve as the starting point for anyone using Docker for the first time.

Comparison to VM Snapshots:

- Think of a Docker image as a snapshot of a virtual machine (VM) environment, capturing everything needed to run the application.

Docker-File

- Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. This page describes the commands you can use in a Dockerfile.

Docker-File-Commands

- ADD Add local or remote files and directories.
- ARG Use build-time variables.
- CMD Specify default commands.
- COPY Copy files and directories.
- ENTRYPOINT Specify default executable.
- ENV Set environment variables.
- EXPOSE Describe which ports your application is listening on.
- FROM Create a new build stage from a base image.
- HEALTHCHECK Check a container's health on startup.
- LABEL Add metadata to an image.
- MAINTAINER Specify the author of an image.
- ONBUILD Specify instructions for when the image is used in a build.
- RUN Execute build commands.
- SHELL Set the default shell of an image.
- STOPSIGNAL Specify the system call signal for exiting a container.

- USER Set user and group ID.
- VOLUME Create volume mounts.
- WORKDIR Change working directory.

#Let's create our 1st Dockerfile

- Clone the ecomm code
- cd ecomm
- vi Dockerfile

#Create a Docker-file

#Use FROM module to select the BASE image
#Use httpd as a BASE image

FROM httpd

#Copy code on httpd
#Use COPY module to copy the code

COPY . /usr/local/apache2/htdocs/

#Now, this dockerfile will help us to build a custom-image
#Build the image

- docker build -t ec-image .

#Let’s create 2nd Docker-File

- cd food-app
- vi Dockerfile

#Use FROM module to select Base image
#Select NGINX as a base image

FROM nginx

#Copy code to NGINX

#USe COPY module to copy the code

COPY . /usr/share/nginx/html

Multi-Stage-Docker-Image

- A **multi-stage Dockerfile** helps you keep your final image small and clean by separating the build environment from the runtime environment. Here's a basic example using a Node.js app

React-App

- Clone the code
- Git clone https://github.com/akracad/Gold_Site.git

###uncomment the multi stage build if you are practising before helm, comment the second docker file#########
#FROM node:18 AS build machine
#WORKDIR /app
#COPY package*.json ./
#RUN npm install
#COPY . .
#RUN npm run build

#FROM nginx:alpine
#COPY --from=build machine /app/build /usr/share/nginx/html
#EXPOSE 80
#CMD ["nginx", "-g", "daemon off;"]

Java - App

-------- Stage 1: Build with Maven --------
FROM maven:3.9.5-eclipse-temurin-17 AS builder

```
# Set the working directory
WORKDIR /app

# Copy pom.xml and download dependencies
COPY pom.xml .
RUN mvn dependency:go-offline

# Copy the source code and build
COPY src ./src
RUN mvn clean package

# -------- Stage 2: Run with Tomcat --------
FROM tomcat:10.1-jdk17

# Remove the default webapps
RUN rm -rf /usr/local/tomcat/webapps/*

# Copy the built WAR file from the builder stage
COPY --from=builder /app/target/*.war /usr/local/tomcat/webapps/ROOT.war

# Expose default Tomcat port
EXPOSE 8080

# Start Tomcat
CMD ["catalina.sh", "run"]
```

Docker Compose

- Docker Compose is a tool for defining and running multi-container applications. It is the key to unlocking a streamlined and efficient development and deployment experience.
- Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single YAML configuration file. Then, with a single command, you create and start all the services from your configuration file.

- Compose works in all environments; production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

Compose-Commands

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Sample-Compose-File

```
version: '3.8'

services:
  app1:
    image: your-app-image
    ports:
      - "8081:80"

  app2:
    image: your-app-image
    ports:
      - "8082:80"

  app3:
    image: your-app-image
    ports:
      - "8083:80"
```

Docker-Compose-Key commands

- To start all the services defined in your compose.yaml file:

```
docker compose up
```

- To stop and remove the running services:

docker compose down

- If you want to monitor the output of your running containers and debug issues, you can view the logs with:

docker compose logs

- To list all the services along with their current status:

- docker compose ps

Docker-Swarm

- Docker Swarm is a container orchestration tool that manages and deploys containerized applications across a cluster of Docker hosts. It simplifies the process of managing, scaling, and maintaining containerized applications in a distributed environment. Docker Swarm allows you to connect containers to multiple hosts, ensuring high availability for your applications.

Key Concepts:

- Swarm: A cluster of Docker hosts (physical or virtual machines) configured to work together.
- Nodes: The individual machines within the Swarm.
- Managers: Nodes that manage the Swarm, including scheduling tasks and handling cluster operations.
- Workers: Nodes that run the containers.
- Services: Unit of deployment that defines the desired state of a containerized application.
- Tasks: Individual container instances that run a service.
- Replicated Services: Services that run multiple instances of a containerized application across the swarm.
- Global Services: Services that run a single instance of a containerized application on each node in the swarm.

How it works:

1. Initialization:

- You initialize a Swarm, typically on one machine acting as the manager.

2. Join Nodes:

- You can then add other Docker hosts as either managers or workers to the Swarm.

3. Service Deployment:

- You deploy your applications as services, defining how many instances should run, where they should be placed, and other configurations.

4. Task Scheduling:

- The managers schedule tasks to run on worker nodes based on the service configurations.

5. Load Balancing:

- Swarm can automatically distribute traffic across multiple services, ensuring high availability.

6. Scaling:

- You can easily scale your services by increasing the number of replicas or adding more nodes to the Swarm.

7. Desired State Reconciliation:

- Swarm continuously monitors the cluster and ensures that the actual state of the containers matches the desired state defined in the services.
- Benefits of using Docker Swarm:
- Simplified Deployment: Easily deploy and manage containerized applications across multiple hosts.
- High Availability: Ensure your applications remain operational even if individual nodes fail.
- Scalability: Easily scale your applications based on demand.

- Load Balancing: Distribute traffic across multiple instances of your application.
- Ease of Management: A centralized management interface for your containerized applications.
- Secure by default: Provides security features for communication within the Swarm.

Universal-Control Plane

- Universal Control Plane is a containerized application that runs on Docker Enterprise Edition, extending its functionality to simplify the deployment, configuration, and monitoring of your applications at scale.
- Once a Universal Control Plane (UCP) instance is deployed, developers and IT operations no longer interact with Docker Engine directly, but interact with UCP instead. Since UCP exposes the standard Docker API, this is all done transparently, so that you can use the tools you already know and love, like the Docker CLI client and Docker Compose.

Features

- Docker Universal Control Plane (UCP) - is the enterprise-grade cluster management solution from Docker. You install it on-premises or in your virtual private cloud, and it helps you manage your Docker cluster and applications through a single interface.

- Centralized cluster management With Docker, you can join up to thousands of physical or virtual machines together to create a container cluster that allows you to deploy your applications at scale. Docker Universal Control Plane extends the functionality provided by Docker to make it easier to manage your cluster from a centralized place.

- Deploy, manage, and monitor With Docker UCP, you can manage from a centralized place all of the computing resources you have available, like nodes, volumes, and networks. You can also deploy and monitor your applications and services.

- Built-in security and access control Docker UCP has its own built-in authentication mechanism and integrates with LDAP services. It also has role-based access control (RBAC), so that you can control who can access and make changes to your cluster and applications. Docker UCP integrates with Docker Trusted Registry so that you can keep the Docker images you use for your applications behind your firewall, where they are

safe and can’t be tampered with. You can also enforce security policies and only allow running applications that use Docker images you know and trust.