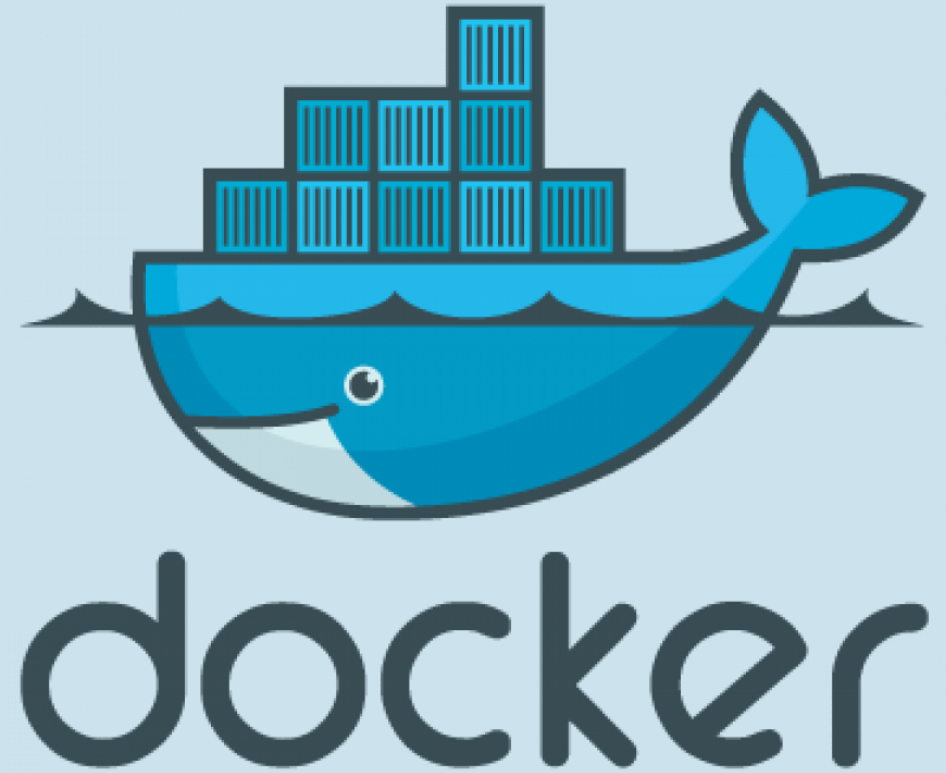


# Get to Docker

## Session 3



# Agenda

- Docker Engine
- Configure resource constraints
- Layered Architecture
- Docker Storage
- Docker Networking

# Docker Engine

- When you install docker, you are actually installing 3 different components
  - The Docker CLI
  - The Docker Engine API
  - The Docker Daemon

# Docker Engine - Docker Daemon

- The Docker Daemon is the main component of the Docker Engine
- It is responsible for
  - Listening for Docker API requests
  - Managing Docker objects such as images, containers, volumes, and networks

# Docker Engine - Docker Engine API

- Docker provides an [API](#) for interacting with the Docker daemon (called the Docker Engine API), as well as SDKs for Go and Python.
- It's a Restful API accessed via HTTP
- You can use the Docker CLI to send requests to the Docker API
- You can also use the Docker SDK to interact with the Docker API using different programming languages

# Docker Engine - Docker CLI



- The Docker CLI is a command-line tool that allows you to interact with the Docker daemon using commands
- It uses the Docker Engine API to interact with the docker daemon
- The Docker CLI need not necessarily be installed on the same machine as the Docker daemon
- You can use the Docker CLI to interact with a remote Docker daemon

## Configure resource constraints



- By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows
- Docker provides ways to control how much memory, or CPU a container can use, setting runtime configuration flags of the `docker run` command
- It achieves this by using the `cgroups` feature of the Linux kernel

# Configure resource constraints

- Ex. Limiting the cpu usage of a container to 50%

```
  ~ docker run -it --cpus=.5 ubuntu bash
```

- Ex. Limiting the memory usage of a container to 1GB

```
  ~ docker run -it --memory=1G ubuntu nginx
```



# Layered Architecture

- Docker uses a layered architecture to store images
- Each line in the Dockerfile creates a new layer in the image
- Example

```
FROM Ubuntu:latest

RUN apt-get update && apt-get install -y python3

RUN pip3 install numpy

COPY src1 /app

CMD [ "python3", "src1/main.py" ]
```

```
FROM Ubuntu:latest

RUN apt-get update && apt-get install -y python3

RUN pip3 install numpy

COPY src2 /app

CMD [ "python3", "src2/main.py" ]
```

- When build the second image, docker will reuse the first 3 layers from the cache and only create the last 2 layers
- This is how docker saves space and time when building images

# Docker Storage

- When first install docker it creates `/var/lib/docker`
- Inside this folder docker stores all its data **by default**
- By data we mean files related to images and containers running on the docker host
- For example
  - all files related to containers are stored under `/var/lib/docker/containers`
  - all files related to images are stored under `/var/lib/docker/image`

# Docker Storage

- When you build an image, the layers of the image become read-only, so you can't modify them
- If you want to modify them you have to initiate a new build
- All containers based on the same image **share** the same read-only layers

# Docker Storage

- When you run a container based of an image, docker create a new read-write layer on top of the read-only layers
- This layer is used to store data created by the container such as
  - Log files
  - Any temporary files generated by the container
  - Any file modified on that container
- This read-write layer is called the **container layer**
- When you stop the container, the container layer and all of the changes stored in it are **deleted**

# Docker Storage

- In the previous docker image, if we run a container based on that image and modify a file in the container (for example the src1/main.py), the file will be copied and the modifications will be stored in the copied version of the file in the container layer
- This is called **copy-on-write**
- By this mechanism, we can have multiple containers running on the same image and each container can have its own copy of the file with its own modifications

# Docker Storage - Volume Mounting

- We said that the container layer is deleted when the container is stopped and all the changes are lost, so how can we persist the changes?
- We can use **volumes**
- Volumes are a way to persist the data generated by the container
- Volumes are stored outside the container layer and are not deleted when the container is stopped

# Docker Storage - Volume Mounting



- You can create a volume using the

```
docker volume create <volume-name>
```

- This will create a new folder under `/var/lib/docker/volumes/<volume-name>`
- You can then mount this volume to a container using the `-v` option, and specifying the name of the volume and the path of the folder on the container

```
docker run -v <volume-name>:<container-path> <image-name>
```

- If the `<volume-name>` does not exist, docker will create it for you
- Example

```
  ~ docker run -v dbVolume1:/var/lib/postgresql/data -d -p 5432:5432 --name db1 -e POSTGRES_PASSWORD=123 postgres
```

# Docker Storage - bind mounting

- You can also mount a folder on the host machine to the container
- You will need to use the `-v` option and specify the path of the folder on the host machine and the path of the folder on the container

```
docker run -v <path-on-host>:<path-on-container> <image-name>
```

- Example

```
docker run -v /home/osc/tmp/dbStorage:/var/lib/postgresql/data -d -p 5432:5432 --name db1 -e POSTGRES_PASSWORD=123 postgres
```



# Docker Storage - Using `--mount` option

- You can also use the `--mount` option to mount volumes and bind mounts
- The `--mount` option is more flexible than the `-v` option
- You can use the `--mount` option to specify the type of the mount, the source of the mount, and the destination of the mount

- **Ex. Volume Mounting**

```
docker run --mount type=volume,source=myVolume,target=/var/lib/postgresql/data -d -p 5432:5432 --name db1 -e POSTGRES_PASSWORD=123 postgres
```

- **Ex. Bind Mounting**

```
docker run --mount type=bind,source=/tmp/dbStorage,target=/var/lib/postgresql/data -d -p 5432:5432 --name db1 -e POSTGRES_PASSWORD=123 postgres
```

# Hands-on

1. Create a container from alpine image with max cpu usage of `50%` and max memory usage of `512m`
2. Create a volume called `myVolume` and mount it to a postgres container using `--mount` option
3. Create a folder called `myFolder` in your home directory and mount it to a postgres container **without** using `--mount` option

Note → the location where postgres stores its data is `/var/lib/postgresql/data`

# Docker Networking

- When you install docker it creates 3 networks by default:
  - bridge
  - none
  - host

```
> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
8aaaab951d63        bridge             bridge              local
771ba4d28add        host               host                local
679ac1f90551        none              null                local
```

- to attach a container to a specific network you can use the `--network` option

```
docker run --network <network-name> <image-name>
```

# Docker Networking - default bridge network

- private internal network created by docker on the host
- all containers attached to this network by default and they get an internal IP address
- containers can communicate with each other using their internal IP addresses
- to access any container inside this network from outside the network you can use the port mapping

```
docker run -p <host-port>:<container-port> <image-name>
```

# Docker Networking - default bridge network

```
> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7368406311ba	alpine	"sh"	5 minutes ago	Up 5 minutes		ali
cfa4123e56fb	alpine	"sh"	6 minutes ago	Up 5 minutes		karim

```
> bridge link
```

```
10: vethea4c10d@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master docker0 state forwarding priority 32 cost 2  
12: vethe91da1c@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master docker0 state forwarding priority 32 cost 2
```

# Docker Networking - default bridge network

```
"ConfigOnly": false,
"Containers": {
  "7368406311ba2a55d57a526b5b20396534366568aaecee7cdebe2a19fdd171d1": {
    "Name": "ali",
    "EndpointID": "d0ca380c7f6876ce520ac14021e5a034cfdb3183394c864157017f924c8720ca",
    "MacAddress": "02:42:ac:11:00:03",
    "IPv4Address": "172.17.0.3/16",
    "IPv6Address": ""
  },
  "cfa4123e56fb29527a423e66c7fd8af9b840bee148c9a6eeb4868d1e04062c85": {
    "Name": "karim",
    "EndpointID": "f913671c488c9e9129a1a646367946473a14c0fd90ef3fa8cf69d3bd6fac9c19",
    "MacAddress": "02:42:ac:11:00:02",
    "IPv4Address": "172.17.0.2/16",
    "IPv6Address": ""
  }
},
}
```

## Docker Networking - none network

- When you run a container with the `--network none` option, the container does not get any network access
- This means that the container cannot communicate with the host or any other containers
- This is useful when you want to run a container in complete isolation
- You can still access the container using the `docker exec` command

## Docker Networking - none network

```
> docker run -it --network=none alpine sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
/ #
```



# Docker Networking - host network

- It is one of the 3 default networks created by docker
- When you run a container with the `--network host` option, the container shares the host's network stack
- This means that the container does not get its own network namespace, and it uses the host's network stack
- This removes the network isolation between the container and the host

# Docker Networking - host network

```
> docker run ahmedyasser9/osc-server  
2024/02/04 10:23:07 Listening on port 7000...
```

```
> curl localhost:7000/osc  
curl: (7) Failed to connect to localhost port 7000 after 0 ms: Couldn't connect to server
```



# Docker Networking - host network

```
> docker run --network=host ahmedyasser9/osc-server  
2024/02/04 10:24:20 Listening on port 7000...
```

```
> docker run --network=host ahmedyasser9/osc-server  
2024/02/04 10:24:23 Listening on port 7000...  
2024/02/04 10:24:23 [server.ListenAndServe, listen tcp :7000: bind: address already in use]
```



# Docker Networking - user-defined bridge network

- You can create your own bridge network using the `docker network create` command

```
docker network create <network-name>
```

- Containers attached to the same network can communicate with each other using their container names or their internal IP addresses

# Docker Networking - user-defined bridge network

```
> docker network create session3
cd5c92ad9f56948b931910c69a88065440bba2efcf9db66d11c20f2bf6484af7
> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
8aaaab951d63        bridge             bridge              local
771ba4d28add        host               host                local
679ac1f90551        none               null                local
cd5c92ad9f56        session3           bridge              local
```

```
> bridge link
17: veth0df1c83@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br-cd5c92ad9f56 state forwarding priority 32 cost 2
19: vethb3134b8@if18: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br-cd5c92ad9f56 state forwarding priority 32 cost 2
```

# Docker Networking - user-defined bridge network

```
"Containers": {  
  "83eaf60b5d28f7e2d6c256c23f4fafed72e9226baa38a830eb097580f1954c0c": {  
    "Name": "ehab",  
    "EndpointID": "e19e3117aad060eda9e87e481a2bc40138b7705a2b599878fddc413f9fde8fa1",  
    "MacAddress": "02:42:ac:14:00:03",  
    "IPv4Address": "172.20.0.3/16",  
    "IPv6Address": ""  
  },  
  "8dc3de810d4e2900b5dc59840c8768be61bbdce445b2d5b8f5a1eb20c30d88bb": {  
    "Name": "nour",  
    "EndpointID": "c0511aff3c28f7387abeab705bc772170a77c1974e97ad7f0d9ec77b71bfbc1e",  
    "MacAddress": "02:42:ac:14:00:02",  
    "IPv4Address": "172.20.0.2/16",  
    "IPv6Address": ""  
  }  
},
```

# Docker Networking - user-defined bridge network

```
> docker attach nour
/ # ping ehab
PING ehab (172.20.0.3): 56 data bytes
64 bytes from 172.20.0.3: seq=0 ttl=64 time=0.205 ms
64 bytes from 172.20.0.3: seq=1 ttl=64 time=0.139 ms
64 bytes from 172.20.0.3: seq=2 ttl=64 time=0.145 ms
^C
--- ehab ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.139/0.163/0.205 ms
/ #
/ #
/ # ping 172.20.0.3
PING 172.20.0.3 (172.20.0.3): 56 data bytes
64 bytes from 172.20.0.3: seq=0 ttl=64 time=0.189 ms
64 bytes from 172.20.0.3: seq=1 ttl=64 time=0.134 ms
64 bytes from 172.20.0.3: seq=2 ttl=64 time=0.133 ms
^C
--- 172.20.0.3 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.133/0.152/0.189 ms
/ # █
```