iNurture Education Solutions
TOMORROW'S HERE

Subject :Containerization using Dockers

Module Number: 4

Module Name: Docker Networking and Docker APIs

Docker Networking and Docker APIs

Syllabus:

- Introduction to Docker Networking.
- None Network.
- Bridge Network.
- Host Network.
- Overlay Network.
- Container Networks with Docker Compose.
- The Docker APIs.
- Engine API.
- Managing images and containers with API.
- Authenticating the Docker Engine API.

Docker Networking and Docker APIs

AIM:

The aim of this module is to learn docker networking and management.

Docker Networking and Docker APIs

Objectives:

- Explain the concepts, functions and components of docker containers.
- Understand the use and benefits of containerization.
- Articulate Docker compose to create multi-container applications.
- Design the best practices of Dockerfiles and Image building.
- To understand how to Run Docker Commands on the command line.
- Create and run Docker containers.

Docker Networking and Docker APIs

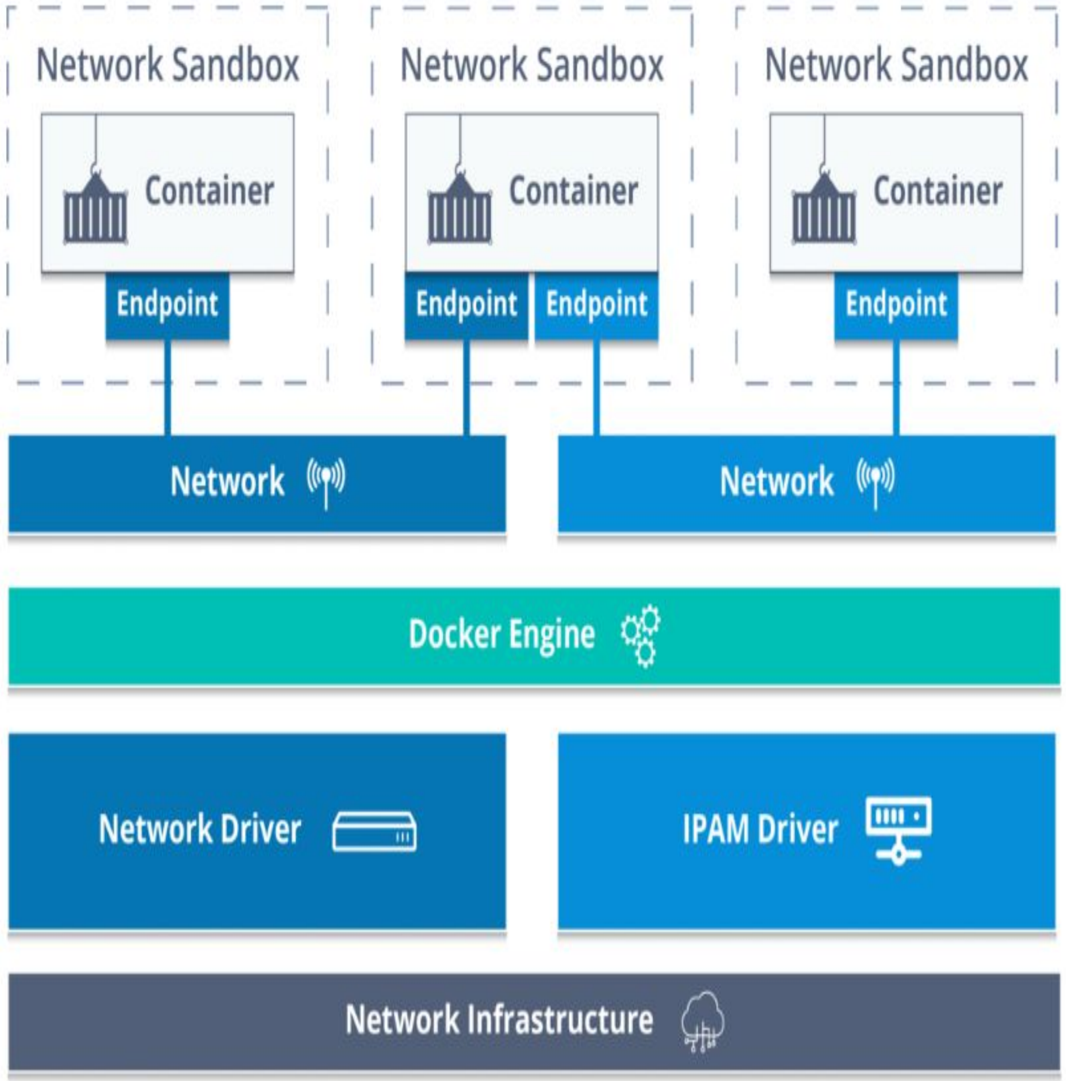
Course Outcomes:

- Develop Containerized applications and implement continuous integration using Docker, explaining different types of cloud deployment and service models.
- Create own images and build the repository.
- Utilize Docker Orchestration and Service discovery features.
- Apply development tools, frameworks, platforms, libraries and packages to test hardware and software systems.
- Evaluate the fundamentals of solution architecture to provision cloud infrastructure.

Docker Networking and Docker APIs

Introduction to Docker Networking

- Containers run in isolation, by default, and don't know anything about other processes or containers on the same machine. So, how do we allow one container to communicate to another? The answer is **networking**.
- If two containers are on the same network, they can talk to each other. If they aren't, they can't.
- Docker networking allows us to attach a container to as many networks as we wish. We can also attach an already running container.



Docker Networking and Docker APIs

Network drivers

Network drivers make Docker’s networking system pluggable. Various drivers exist by default and provide core networking functionality:

- Bridge
- Host
- Overlay
- Macvlan
- None

Docker Networking and Docker APIs

Bridge Network

- A bridge network makes use of a software bridge that allows containers connected to the same bridge network to communicate with each other, while at the same time, providing isolation from containers which are not connected to that bridge network.
- The Docker bridge network driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other.
- Bridge is the default network driver. If you don't specify any driver, this is the type of network that gets created.
- Bridge networks are generally preferred when the applications run in standalone containers which need to communicate.
- User-defined custom bridge networks are best when you need multiple containers to communicate on the same Docker host.

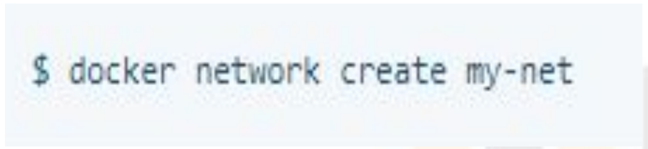
Difference between user-defined bridge and default bridge

- User-defined bridges provide automatic DNS resolution between containers.
- User-defined bridges provide better isolation.
- Containers can be attached and detached from user-defined networks on the fly.
- Each user-defined network creates a configurable bridge.

Docker Networking and Docker APIs

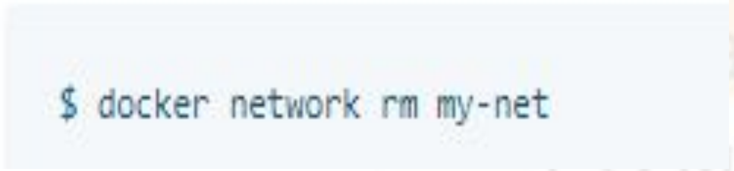
Managing user-defined bridge

- User-defined bridge networks can be created using following command:



```
$ docker network create my-net
```

- We can specify the subnet, IP address range, gateway and other required options.
- To remove a user-defined bridge network, use following command:



```
$ docker network rm my-net
```

- If containers are currently connected to the network, they have to be disconnected first.

Docker Networking and Docker APIs

Managing user-defined bridge

- Another example: create a bridge network-

```
$ docker network create -d bridge my_bridge
```

- The –d flag tells docker to use bridge driver for new network. We could leave this flag off as bridge is the default value for this flag.
- We can even list the networks on the machine as below:

```
$ docker network ls

NETWORK ID          NAME                DRIVER
7b369448dccb        bridge              bridge
615d565d498c        my_bridge           bridge
18a2866682b8        none                null
c288470c46f6        host                host
```

Docker Networking and Docker APIs

Host network

- If host network mode is used for a container, that container's network stack is not isolated from the Docker host i.e. the container shares the host's networking namespace and does not have its own IP-address allocated.
- For example, if we use host networking and run a container which binds to port 80, then the container's application is available on port 80 on the host's IP address.
- Note: Since the container does not get its own IP-address while on host networking mode, port-mapping does not come into picture.
- Host mode networking helps in optimizing performance and in scenarios where a container needs to handle a huge range of ports, as it does not need to go for network address translation (NAT).
- The host network driver works only on Linux hosts. It is not supported on Docker Desktop for Mac, Windows or Docker EE for Windows Server.
- Thus **Host networks** are best suited for situations where network stack need not be isolated from the Docker host, but one wishes that other aspects of the container be isolated.

Docker Networking and Docker APIs

Overlay network

- Overlay networks link multiple Docker daemons together and enable swarm services to communicate with each other.
- The overlay network driver creates a distributed network among multiple Docker daemon hosts. This network sits on top of (overlays) the host-specific networks, letting containers connected to it, to communicate securely when encryption is enabled. Docker transparently takes care of routing each packet to and from the correct Docker daemon host and the correct destination container.
- We can use overlay networks to facilitate communication between a swarm service and a standalone container or between two standalone containers on different Docker daemons. This strategy rules out the need for OS-level routing between containers.
- Thus **Overlay networks** are best suited for situations wherein we need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.

Docker Networking and Docker APIs

Overlay network

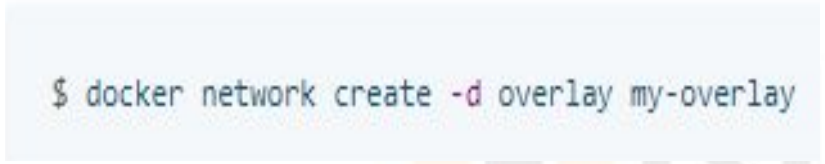
When we initialize a swarm or when a Docker host joins an existing swarm, two new networks are created on that Docker host, namely:

- ***Overlay network called ingress***: controls and manages data traffic related to swarm services. When we create a swarm service and do not connect it to a user-defined overlay network, then by default, it connects to the ingress network.
- ***Bridge network called docker_gwbridge***: connects the individual Docker daemon to other daemons participating in the swarm.

Docker Networking and Docker APIs

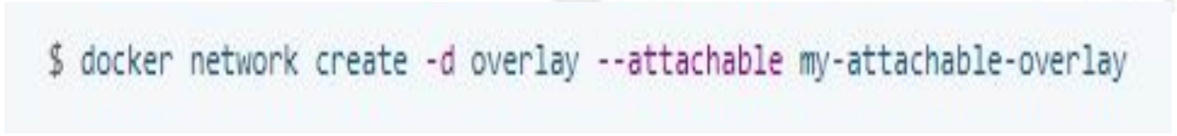
Create an overlay network

- To create an overlay network in order to use with swarm services, use following command:



```
$ docker network create -d overlay my-overlay
```

- To create an overlay network that can be used by swarm services **or** standalone containers to communicate with other standalone containers which run on other Docker daemons, add the --attachable flag as in below command:



```
$ docker network create -d overlay --attachable my-attachable-overlay
```

Docker Networking and Docker APIs

Macvlan network

- Macvlan networks assign a MAC address to a container, making it look like a physical device on the network. The Docker daemon routes traffic to containers by their MAC addresses.
- Certain legacy applications or those applications which monitor network traffic, expect to be directly connected to the physical network, rather than routed through the Docker host's network stack. Opting for macvlan network driver would be best while dealing with such applications.
- The macvlan network driver assigns a MAC address to each container's virtual network interface, making it look like a physical network interface connected directly to the physical network.

Keep following things in mind:

- Unintentionally damage the network is easy due to IP address exhaustion or "VLAN spread", which is a scenario in which we have inappropriately large number of unique MAC addresses in the network.
- Our networking equipment should be able to handle "promiscuous mode", where one physical interface can be assigned multiple MAC addresses.
- If an application can work using a bridge (on a single Docker host) or overlay (to communicate across multiple Docker hosts), these solutions may be better in the long term.
- **Macvlan networks** are best suited while migrating from a VM setup or when containers need to look like physical hosts on network, each with a unique MAC address.

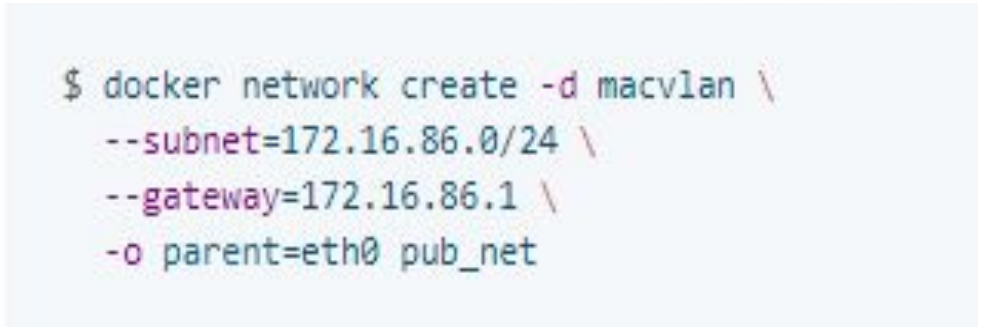
Docker Networking and Docker APIs

Create a Macvlan network

When a macvlan network is created, it can be either in bridge mode or 802.1q trunk bridge mode.

- In bridge mode, macvlan traffic goes through a physical device on the host.
- In 802.1q trunk bridge mode, traffic goes through an 802.1q sub-interface which Docker creates on the fly. This allows to control routing and filtering at a more granular level.

(1) BRIDGE MODE: To create a macvlan network that bridges with a given physical network interface, use --driver macvlan with the docker network create command. We also need to specify the parent, which is the interface that the traffic will physically go through on the Docker host.

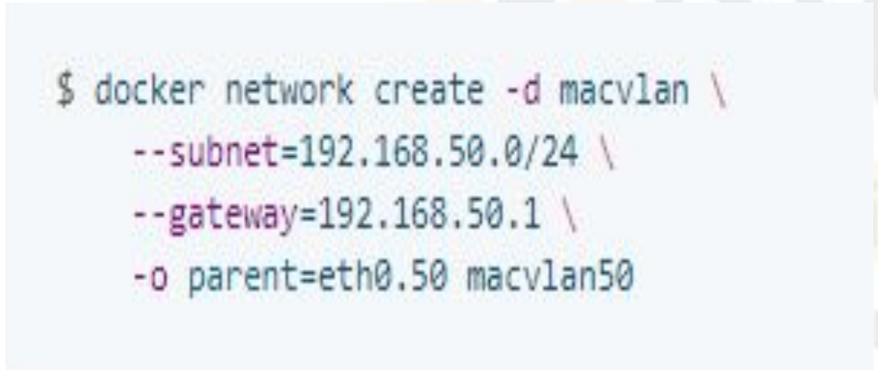


```
$ docker network create -d macvlan \
  --subnet=172.16.86.0/24 \
  --gateway=172.16.86.1 \
  -o parent=eth0 pub_net
```

Docker Networking and Docker APIs

Create a Macvlan network

(2) 802.1q TRUNK MODE: If a parent interface name is specified with a dot included, such as eth0.50, Docker interprets that as a sub-interface of eth0 and creates the sub-interface automatically.



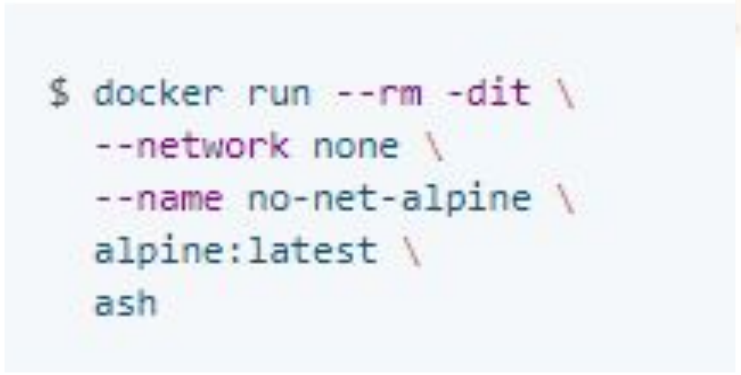
```
$ docker network create -d macvlan \
    --subnet=192.168.50.0/24 \
    --gateway=192.168.50.1 \
    -o parent=eth0.50 macvlan50
```

Docker Networking and Docker APIs

None network

- Disable all networking for the container. It is generally used along with a custom network driver.
- none is not available for swarm services.
- If we want to totally disable the networking stack on a container, then --network none flag can be used while starting the container. This is shown in following example:

1. Create the container



```
$ docker run --rm -dit \
  --network none \
  --name no-net-alpine \
  alpine:latest \
  ash
```

Docker Networking and Docker APIs

None network

2. Check the container's network stack, by executing some common networking commands within the container. Notice that no eth0 was created.

```
$ docker exec no-net-alpine ip link show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN qlen 1
    link/ipip 0.0.0.0 brd 0.0.0.0
3: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop state DOWN qlen 1
    link/tunnel6 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00 brd 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
```

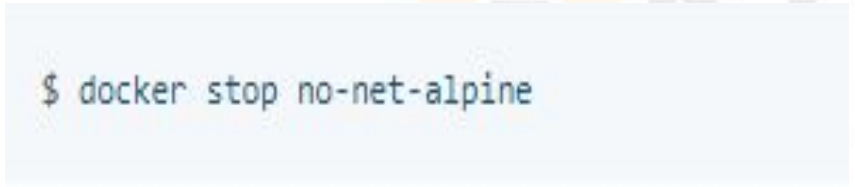
The below command returns empty because there is no routing table.

```
$ docker exec no-net-alpine ip route
```

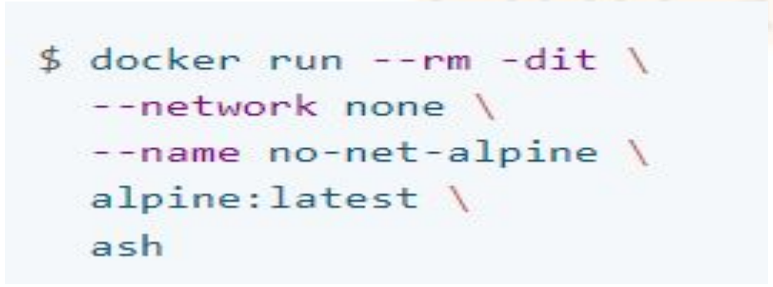
Docker Networking and Docker APIs

None network

3. Stop the container. It is removed automatically as it was created with the --rm flag



```
$ docker stop no-net-alpine
```



```
$ docker run --rm -dit \
  --network none \
  --name no-net-alpine \
  alpine:latest \
  ash
```

Docker Networking and Docker APIs

Container networks with Docker Compose

- Docker Compose is primarily a tool meant for defining and running multi-container Docker applications.

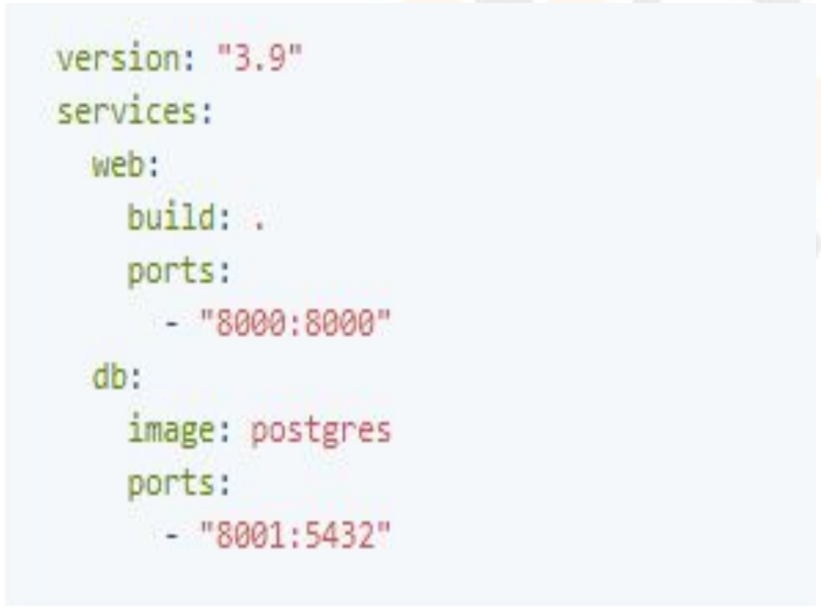
<u>Using Compose is a three-step process</u>:

- First, define your app's environment with a *Dockerfile* so that it can be reproduced anywhere.
- Next, define and configure the services which form our app in *docker-compose.yml* so that they can be run together in an isolated environment.
- Run *docker-compose up* and then Compose starts and runs the entire application.

Docker Networking and Docker APIs

Container networks with Docker Compose

- Compose sets up a single network for our app by default. Each container for an application service joins the default network.
- It is reachable by other containers on that network and also discoverable at a hostname which is identical to the container name.
- For example, our app is in a directory called myapp, and our docker-compose.yml is as follows:



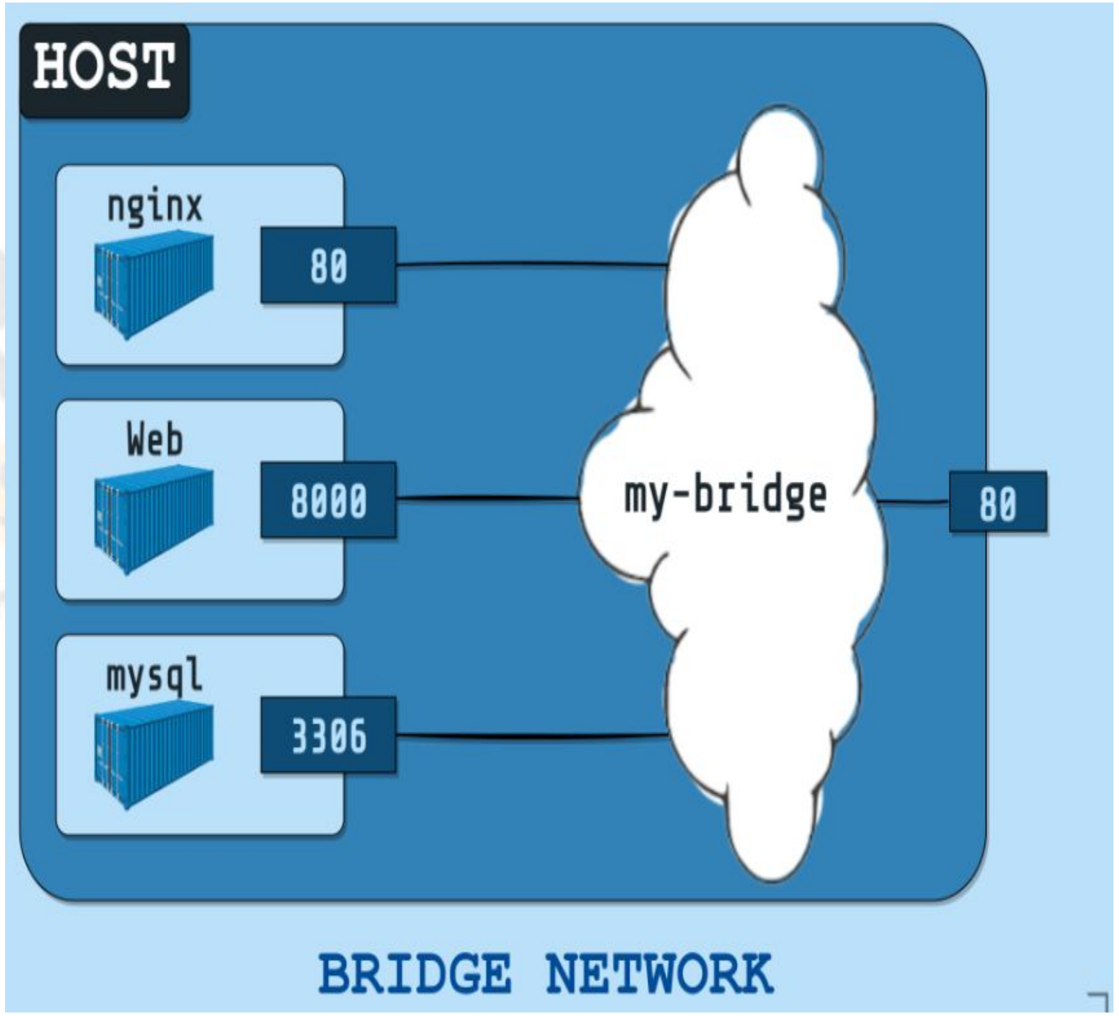
```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
    ports:
      - "8001:5432"
```

Docker Networking and Docker APIs

Container networks with Docker Compose

When we run docker-compose, the following happens:

- A network called myapp_default is created.
- A container is created using web's configuration. It joins the network myapp_default under the name web.
- A container is created using db's configuration. It joins the network myapp_default under the name db.



Docker Networking and Docker APIs

Container networks with Docker Compose

- Then each container looks up the hostname web or db and gets back the container's IP address appropriately. For example, web's application code connects to the URL postgres://db:5432 and begins using the Postgres database.
- It is essential to note the difference between HOST_PORT and CONTAINER_PORT. In above example, HOST_PORT is 8001 for db and container port is 5432 (postgres default). Networked service-to-service communication uses the CONTAINER_PORT. On defining HOST_PORT, the service is accessible outside the swarm as well.
- Within the web container, our connection string to db would look like postgres://db:5432, and from the host machine, the connection string would look like postgres://{DOCKER_IP}:8001.

Src: https://docs.docker.com/compose/networking/

Docker Networking and Docker APIs

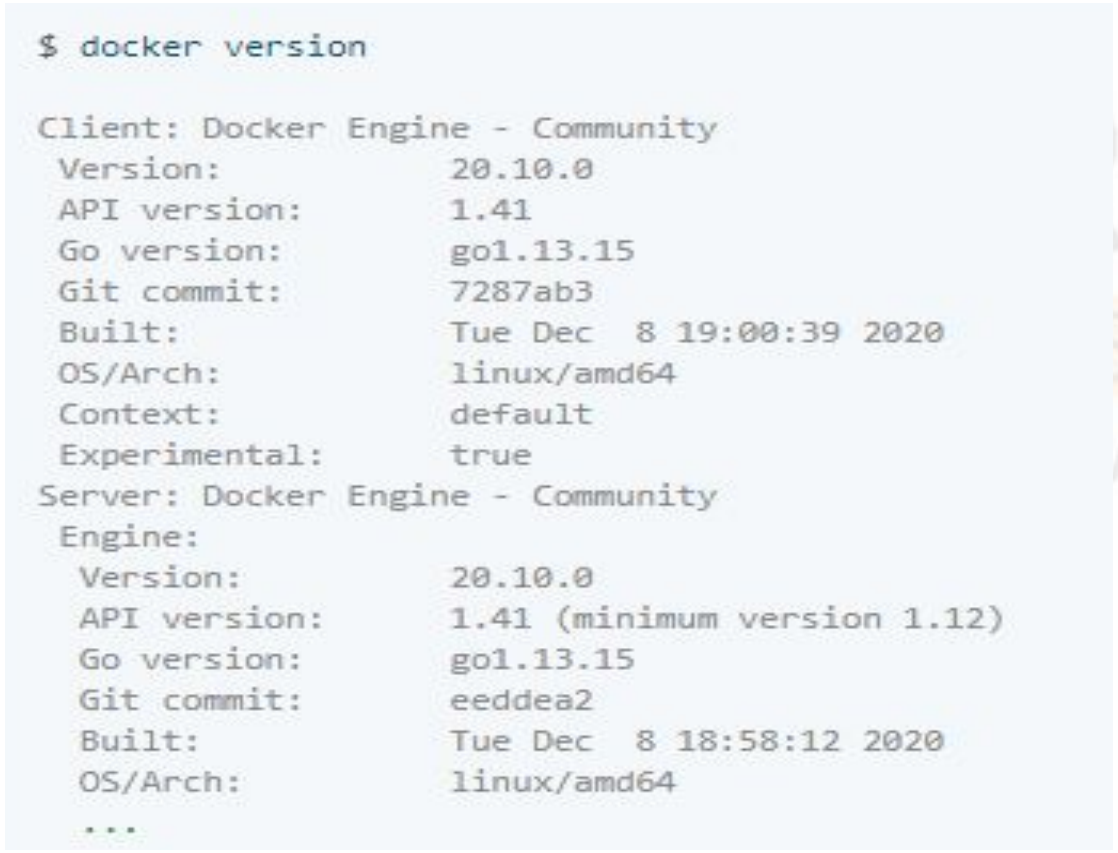
Docker Engine API

- Docker offers **an API to interact with the Docker daemon** (called the Docker Engine API), as well as SDKs for Go and Python. The SDKs help in building and scaling Docker apps and solutions in a quick and easy manner.
- The Docker Engine API is a RESTful API that is accessible by an HTTP client such as wget or curl, or the HTTP library which is part of almost many modern programming languages.
- The version of Docker Engine API relies on the version of Docker daemon and Docker client.
- A given version of Docker Engine SDK supports a specific version of Docker Engine API along with all along previous versions.
- When new features are added, a new version of the API is released.

Docker Networking and Docker APIs

Docker Engine API

- To view the highest version of API supported by a Docker daemon and client, use comand- docker version as below:



```
$ docker version

Client: Docker Engine - Community
 Version:           20.10.0
 API version:       1.41
 Go version:        go1.13.15
 Git commit:        7287ab3
 Built:             Tue Dec  8 19:00:39 2020
 OS/Arch:           linux/amd64
 Context:           default
 Experimental:      true
Server: Docker Engine - Community
 Engine:
  Version:          20.10.0
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       eeddea2
  Built:            Tue Dec  8 18:58:12 2020
  OS/Arch:          linux/amd64
  ...
```

Docker Networking and Docker APIs

Docker Engine API

The API version to use can be specified in one of the following ways:

- While using the SDK, use the latest version you can, but at least the version that incorporates the API version with the features you require.
- When using curl directly, specify the version as the first part of the URL. For instance, if the endpoint is /containers/, you can use /v1.41/containers/.
- In order to force the Docker CLI or the Docker Engine SDKs to use an old version of the API than the version currently reported by docker version, set the environment variable DOCKER_API_VERSION to the correct version. This applies on Linux, Windows, or macOS clients. For example:

```
$ DOCKER_API_VERSION='1.41'
```

Docker Networking and Docker APIs

Docker Engine API

- Upon setting environment variable, the specified old version of API is used, even if Docker daemon supports a newer version. This environment variable disables API version negotiation, and it should only be used if it is mandatory for us to use a specific version of the API, or for debugging purposes.
- The Docker Go SDK allows enabling API version negotiation, by automatically selecting an API version that is supported by both client and Docker Engine that is used.

Managing images and containers with API

- The Engine API is an HTTP API served by Docker Engine. It is used by Docker client to communicate with the Engine.
- Standard HTTP status codes are used by the API to highlight the success or failure of the API call.
- With each release, the API is generally modified, thus API calls are versioned to make sure that clients don't break.
- To lock a particular version of the API, we can prefix the URL with its version, for example, we can call /v1.30/info to use the v1.30 version of the /info endpoint.
- If the daemon does not support the API version specified in the URL, it returns a HTTP 400 Bad Request error message.
- If we avoid the version-prefix, the current version of the API (v1.41) is used.
- Consider, for example, calling /info is the same as calling /v1.41/info.

Docker Networking and Docker APIs

Authenticating Docker Engine API

- Authentication for registries is taken care at client side. The client sends authentication information to different endpoints that need to communicate with registries, such as POST /images/(name)/push. These are sent as X-Registry-Auth header as a base64url encoded (JSON) string with the following structure:

```
{
  "username": "string",
  "password": "string",
  "email": "string",
  "serveraddress": "string"
}
```

- The serveraddress is a domain/IP without a protocol. Double quotes are required throughout this structure.
- If an identity token is already received from the /auth endpoint, we can just pass this instead of credentials:

```
{
  "identitytoken": "9cbaf023786cd7..."
}
```

Docker Networking and Docker APIs

Summary

- Docker networking allows us to attach a container to as many networks as we wish. We can also attach an already running container.
- A bridge network makes use of a software bridge that allows containers connected to the same bridge network to communicate with each other, while at the same time, providing isolation from containers which are not connected to that bridge network. Bridge is the default network driver. If you don’t specify any driver, this is the type of network that gets created.
- Host networks are best suited for situations where network stack need not be isolated from the Docker host, but one wishes that other aspects of the container be isolated.
- Overlay networks are best suited for situations wherein we need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.

Docker Networking and Docker APIs

Summary

- Docker Compose is primarily a tool meant for defining and running multi-container Docker applications.
- Compose sets up a single network for our app by default. Each container for an application service joins the default network.It is *reachable* by other containers on that network and also *discoverable* at a hostname which is identical to the container name.
- Docker offers an API to interact with the Docker daemon (called the Docker Engine API), as well as SDKs for Go and Python. The SDKs help in building and scaling Docker apps and solutions in a quick and easy manner.
- The version of Docker Engine API relies on the version of Docker daemon and Docker client.
- A given version of Docker Engine SDK supports a specific version of Docker Engine API along with all along previous versions.When new features are added, a new version of the API is released.
- Macvlan networks assign a MAC address to a container, making it look like a physical device on the network. The Docker daemon routes traffic to containers by their MAC addresses.
- If we want to totally disable the networking stack on a container, then --network none flag can be used while starting the container.

Docker Networking and Docker APIs

Self Assessment Questions

1. Which is the default network driver for containers?

   a. Host

   b. Bridge

   c. Overlay

   d. None

Answer: b

Docker Networking and Docker APIs

Self Assessment Questions

2. State True or False:

   Docker networking allows us to attach a container to as many networks as we want.

   a. True

   b. False

Answer: a

Self Assessment Questions

3. In which network mode, the container's network stack is not isolated from the Docker host ?

   a. None

   b. Bridge

   c. Host

   d. Overlay

Answer: c

Self Assessment Questions

4. Which networks link multiple Docker daemons together ?

   a. Bridge

   b. Overlay

   c. Macvlan

   d. None

Answer: b

Docker Networking and Docker APIs

Self Assessment Questions

5. __________ is a tool for defining and running multi-container Docker applications.

a. Docker swarm

b. Docker compose

c. Docker hub

d. None

Answer: b

Docker Networking and Docker APIs

Self Assessment Questions

6. The version of Docker Engine API depends on the version of ______________.

a. Docker daemon

b. Docker client

c. Docker daemon and Docker client

d. none

Answer: c

Docker Networking and Docker APIs

Self Assessment Questions

7. State True or False: A given version of Docker Engine SDK does not support all earlier versions of Docker Engine API.

   a. True

   b. False

Answer: b

Docker Networking and Docker APIs

Self Assessment Questions

8. If the daemon does not support the API version specified in the URL, it returns__________ error message.

   a. HTTP 400 Bad Request

   b. 404 Bad Request

   c. Server 402 Wrong Request

   d. HTTP 404 Server error

Answer: a

Docker Networking and Docker APIs

Self Assessment Questions

9. In case of Docker engine APIs, Authentication for registries is handled at:

a. Server side

b. Client side

c. Both

Answer: b

Docker Networking and Docker APIs

Assignment

1. Explain various network drivers which provide docker networking functionality.

2. Write a note on Docker Engine API.

3. Explore and discuss managing images and containers with API(maximum api version 1.41).

   One can refer link https://docs.docker.com/engine/api/v1.41/

Docker Networking and Docker APIs

Video Links

Topics	Video Link	Note
Docker networking, Container network model, various network drivers, demo	https://www.youtube.com/watch?v=c6Ord0GAOp8	Networking Concepts in Docker, explained various networking drivers
Docker Engine API	https://www.youtube.com/watch?v=a5td09OWFXA	Docker Engine and API Which are used
Docker compose networking basics	https://youtu.be/VbaQBIQjRb8	How to create Docker Compose file and run commands

Docker Networking and Docker APIs

E-Book Links and References

Topics	Video Link
Containers and Networking	https://docs.docker.com/engine/tutorials/networkingcontainers/
Docker Image on windows	https://docs.docker.com/engine/api/
Port Configuration and Networking	https://docs.docker.com/network/