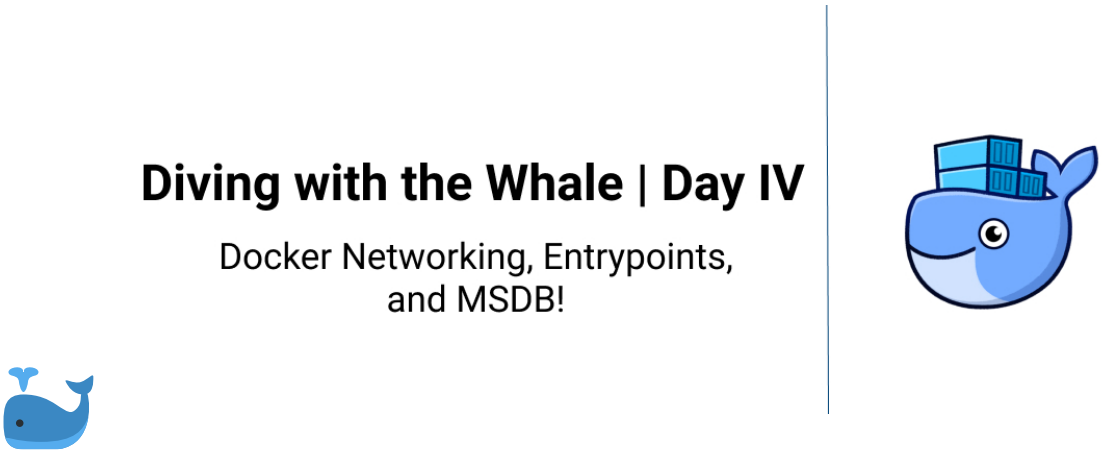


Diving with the Whale - Docker Day IV - Docker Networking, Entry Points, and more.

Author : Ahmed Ayman

Docker Networking

How can we let two or more containers talking to each other ? through docker containers!

Networking in containers is controlling who can talk to who including all the running containers, and the local-host.

▼ **Docker Network Drivers**

Several drivers exist by default in docker, and provide core networking functionality

1. **Bridge Network**

   The default network driver. If you don’t specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate.

   you can also create your own user-defined bridge network and connect multiple container to it so they can contact with each other! - we'll see how in the next section -

   > A network bridge is a computer networking hardware device or a software device running within a host machine’s kernel. that creates a single, aggregate network from multiple communication networks or network segments. This function is called network bridging. Bridging is distinct from routing. Routing allows multiple networks to communicate independently and yet remain separate, whereas bridging connects two separate networks as if they were a single network. In the OSI model, bridging is performed in the data link layer (layer 2). If one or more segments of the bridged

> network are wireless, the device is known as a wireless bridge.
> - Bridging (networking) - Wikipedia

> In terms of Docker, a bridge network uses a software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network. The Docker bridge driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other.
> - Use bridge networks - Docker Docs

2. **Host Network**

   For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly.

   NO PORT MAPPING REQUIRED!

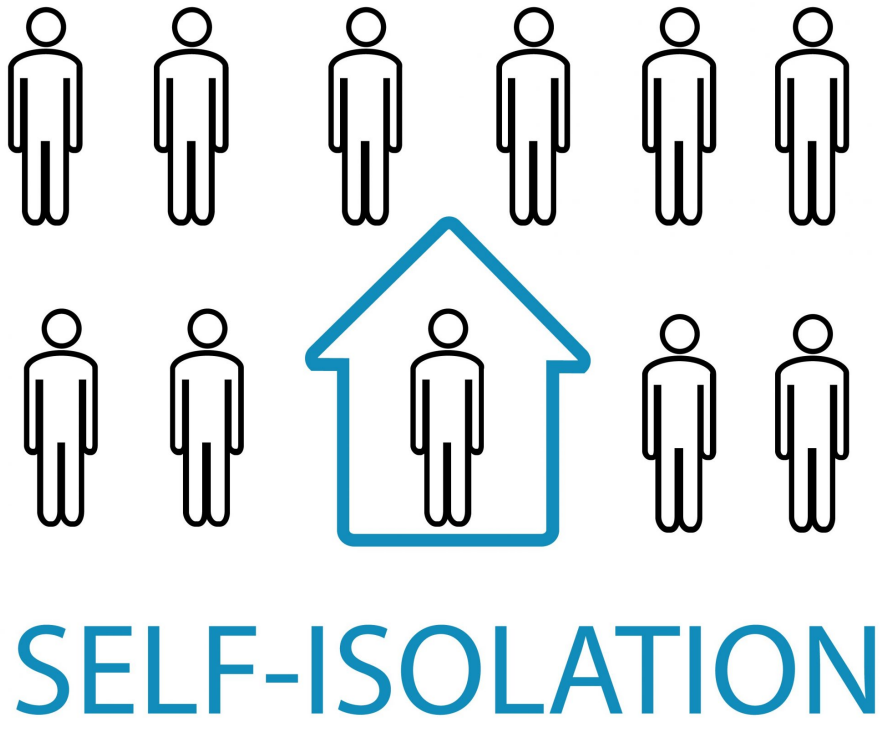
   > If you use the host network mode for a container, that container's network stack is not isolated from the Docker host (the container shares the host's networking namespace), and the container does not get its own IP-address allocated. For instance, if you run a container which binds to port 80 and you use host networking, the container's application is available on port 80 on the host's IP address.
   > - Use host networks - Docker Docs

   for the container running using the host network, any port mapping option will be ignored.

3. **None Network**

For this container, disable all networking. Usually used in conjunction with a custom network driver.

This container will be isolated from any network, it cannot contact with any other machine, or container. It's like the container is in quarantine alone!



4. **Overlay Network**

> Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also 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 removes the need to do OS-level routing between these containers. See overlay networks.
> - Networking Overview - Docker Docs

5. **Macvlan Network**

> Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to

> containers by their MAC addresses. Using the Macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack. See Macvlan networks.
> - Networking Overview - Docker Docs

▼ **Docker Networks Management**

List all Networks

```
docker network ls
```

Lists all the networks along side with there names, ids, drivers, and scope.

Create new Network

```
docker network create $network_name -d $driver
```

Creates a new network with the specified unique name, the default network driver is **bridge**

Remove a Network

```
docker network rm $network_id
```

Remove all Networks

```
docker network prune
```

Get Network Info - Inspect

```
docker network inspect $network_name
```

Gives you a JSON file contains all sort of info about the network, containing the IP address and the Gate-way IP address

Connect a network to a running container

```
docker network connect $network_id $container_id
```

Dis-Connect a network to a running container

```
docker network disconnect $network_id $container_id
```

▼ **Attach a Container to a Network**

When running the Container

```
docker run --network=$network_name $image_name
# for example
docker create webappNetwork -d bridge
docker run -td --network=webappNetwork a7medaymamn6/hello-world
```

Now get the IP address of the network you created

```
docker network inspect web | grep "IPv4" | cut -d ":" -f 2 | cut -d "\""  -f 2 | cut -d "/" -f 1
# 172.18.0.2
```

this will get you the IP address after cutting out the label, the column ':', the double-quotes ' " ', and finally cutting out the net-mask

you can visit this IP with the suffix :5000 adding the port that the web-app working on the browser, and wallah! the website is there!

▼ **Communicate Between two Containers in the same network**

Let's try the thing out, we'll run two containers and ping each other using their names and IPs
We'll use the alpine image which have ping installed by default and it's light

```
# create a new network
docker network create connect-containers
# - 77dcad281962e75132a0aa83028bfa6d632cc09a19d76246d2e68c299c879700

# create the first container
docker run --rm -it --network=connect-containers --name=cont1 alpine
# - now we have a shell back !
ping cont2
# - PING cont1 (172.18.0.2): 56 data bytes
# - 64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.292 ms
# - 64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.167 ms
# - 64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.175 ms
^C
# it's sending the packets which means cont2 is app and running

# let's try to do the same with the ip address
# get the ip address
ifconfig | grep inet
# - inet addr:172.18.0.3  Bcast:172.18.255.255  Mask:255.255.0.0
# - inet addr:127.0.0.1  Mask:255.0.0.0
# 172.18.0.3 this is cont2 (this shell session) ip address
# and 172.18.0.3 is cont1 ip address
ping 172.18.0.2
# - PING 172.18.0.2 (172.18.0.2): 56 data bytes
# - 64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.292 ms
# - 64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.167 ms
```

```
# create the second container
docker run --rm -it --network=connect-containers --name=
# - now we have a shell back !
ping cont1
# - PING cont1 (172.18.0.3): 56 data bytes
# - 64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.292 ms
# - 64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.167 ms
# - 64 bytes from 172.18.0.3: seq=2 ttl=64 time=0.175 ms
^C
# it's sending the packets which means cont1 is app and

# let's try to do the same with the ip address
# get the ip address
ifconfig | grep inet
# - inet addr:172.18.0.2  Bcast:172.18.255.255  Mask:255
# - inet addr:127.0.0.1  Mask:255.0.0.0
# 172.18.0.2 this is cont2 (this shell session) ip addr
# and 172.18.0.3 is cont1 ip address
ping 172.18.0.3
# - PING 172.18.0.3 (172.18.0.3): 56 data bytes
# - 64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.292 ms
# - 64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.167 ms
# - 64 bytes from 172.18.0.3: seq=2 ttl=64 time=0.175 ms
^C
# it's sending the packets which means 172.18.0.3 is ap
```

```
# - 64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.175 ms
^C
# it's sending the packets which means cont2 is app and running
```

CMD vs Entry-points

▼ **CMD**

ONLY ONE CMD COMMAND

Actually I lied, you can have more than one CMD command, but only the last one will get executed!

Sets defaults for running a container

it tells the engine exactly how to run a container from this image if no commands are specified (if any commands were specified it will overwrite the CMD command)

```
CMD ["python3", "app.py"]
# OR
CMD python3 app.py
```

- NOTE : A CMD command in the docker run command will override the default in the Dockerfile.

```
docker run --rm -it ubuntu ls
# bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
# boot  etc  lib   lib64  media opt  root  sbin  sys  usr
# ____________________________________________________
# so the ls command will override the CMD command that written in ubuntu dockerfile which is /bin/bash
# so actually when you run this command you will get the output of ls command and you will not get a shell!
# the ls process gets executed then the container exits
```

▼ **Entry-Point**

In the Dockerfile you can use the entry point to do the same job as CMD, to set the executable that runs whenever the container runs.

```
ENTRYPOINT python3 app.py
```

So what happens when we have an entry point AND a CMD command ?!

in this situation CMD acts as an extension for the command in the ENTRYPOINT, it gets appended to it!
it's like the actual command is the concatenation of ENTRYPOINT_VALUE + CMD_VALUES

Let's try it out, we'll setup a little image based on Ubuntu

```
# build an image on the top of ubuntu image
FROM ubuntu
# exectue the command ls once the container gets run
ENTRYPOINT ls
# append to the ENTRYPOINT command the -a argument
CMD -a
# now the command that will get exectued is ls -a
```

We know that the extra command when executing the run command overwrites the CMD command
so what happens when we have an Entry-point like the above example? let's se

```
docker build -t demo -f Dockerfile
docker run demo -l
```

actually this well overwrite the -a and executes ls -l command!

In the end it's okay to use either or both, most of the times you can get-away with CMD only.

Multi-Stage Docker Builds (MSDB)

▼ **What is MSDB ?**

> MSDB allow you to extract artifacts from docker image builds, leaving behind all the extra junk you don’t need.
> - Mastermnd

▼ **Multiple FROM Commands**

Remember when I said you can only have one FROM command in your Dockerfile ? guess what ! I lied again, get used to it!

Each new FROM command initiates a new stage of the build

You can name the stage using the AS keyword in the FROM command

Why would you do that ?
good question, so here is how the MSDB works, first it builds the first stage complete it, then **REMOVES THE INTERMEDIATE CONTAINER** after that docker starts to build the next stage without the earlier stage!
think about it, let's say you're dockeraizing a C/C++ application for example based on Ubuntu image, the docker file instructions will be smth like this

```
FROM ubuntu AS compile-step
RUN apt update
RUN mkdir /build
WORKDIR /build
COPY prog.c .
RUN gcc -o prog.o prog.c

CMD ./prog.o
```

Read the above Dockerfile thoroughly and think about it!

what do you really need from this container ?
1. to compile and build the program
2. to run the program

so we don't need all the size that comes with the Ubuntu image, we can use it as an intermediate container to compile and build the program, then copy the .o (binary) file into a new fresh container with a smaller size just enough to run

the binary executable file!

here is what our Dockerfile should look like

```
# Stage 1 - compile and build the c program using gcc based on Ubuntu image
FROM ubuntu AS compile-step
RUN apt update
RUN mkdir /build
WORKDIR /build
COPY prog.c .
RUN gcc -o prog.o prog.c

# Stage 2 - run the executable file based on alpine image which is a very small size image
FROM alpine AS run-step
RUN mkdir /app
WORKDIR /app
# copy the binary file from the previous stage
COPY --from=compile-step /build/prog.o .
CMD ./prog.o
```

NOTE : IN THIS EXAMPLE WE COULD USE ALPINE DIRECTLY BUT WE DID THIS FOR THE SAKE OF DEMONSTRATION, BUT TRY TO THINK BIGGER AND MORE COMPLEX SITUATIONS!

Now the actual size of the image will drop down a lot!

alpine image is a very small image with size of 5.61 MB only, and the Ubuntu:latest image size is 72.7 MB

So the big catch here is reducing the size of the image by extracting only what we need into a lighter base image and using the first one to make a certain job then throw it away!