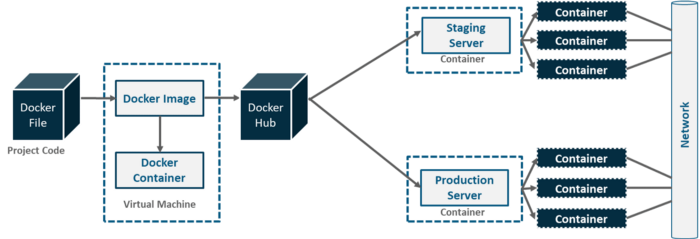
Docker Network



1. Run an nginx and an httpd container.

$ docker container run -d httpd   
  
$ docker container run -d nginx

1. Start a bash session inside the apache container.

docker exec -it <container\_id> bash

Install iputils-ping to try to talk to nginx container

# apt update && apt install iputils-ping -y

 Let’s find out the ip address of the nginx container

docker container inspect <container\_id>

ping nginx container using the ip address we saw above

ping <ip\_address>

we are able talk to the nginx container.

This is possible because of docker networks.

## **types of network**

Docker network has 4 types:

1. bridge
2. host
3. none
4. overlay

Let’s view all the networks present on docker

$ docker network ls

By default every container uses bridge network unless explicitly specified.

1. You can inspect the bridge network to see all containers that are a part of it.

$ docker network inspect bridge

You can see both apache and nginx containers as a part of the default bridge network.

*We will also see that containers use a different ip from that of the host.*

## **Creating our own network**

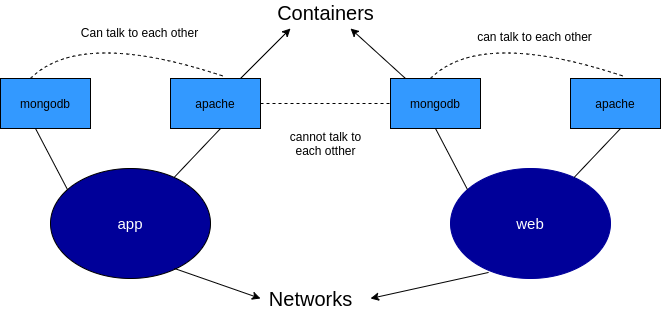
Now, we already know that containers can talk to each other on the same network.

Let’s consider a scenario where we have two different apps running, which are not related to each other. How would we ensure no communication should be allowed between containers of different apps in such a scenario?

Let’s take a look at the solution.

We can create our own networks to isolate containers logically.

*Containers in the same network can talk to each other.*



1. Create a network called network1 and another called network2.
2. docker network ls
3. Start an apache container and add it to the network1

docker container run -d --network network1 httpd

1. Start a mongo db container and add it to app network as well

$ docker container run -d --name db --network network2 mongo

1. Start yet another mongo db container and add it to web network as follows

$ docker container run -d --name webdb --network network2 mongo

1. Start a bash session inside the apache container
2. docker exec -it <container\_id> bash

apt update && apt install iputils-ping -y

Find out the ip address of db container

$ docker container inspect container\_id

ping mongo container using the ip address we saw above

we are able talk to the mongo container in the same network.

 Now, let’s try to ping webdb container of other network. Find out the ip address the same way we did before.

Thus we can ensure containers from different apps can’t talk to each other.

## **How does the outside world interact with the container then?**

This can be achieved by opening ports of the container and map them to the host’s port.

1. Run an nginx container with and map port 80 of the host to port 80 of the container.

$ docker container run -d -p 80:80 nginx

Let’s see if mapping was done successfully

$ docker container port <container\_id>

## **None network**

When you specify a container to use none network it would be completely isolated. It won’t be able to talk to any other container nor the internet.

*No ip address is assigned to such a container.*

$ docker container run -d --network none nginx

Let’s try to log the ip address of this container

$ docker container inspect <container\_id>

You will see there is no ip address assigned to this container.

## **Host network**

Sometimes there is a need to use the host network itself. In this case you lose a few containerisation benefits but networking is fast.

*In this case no new network nor a new ip address assigned. The container uses the host ip itself.*

**Note :**The host networking driver only works on Linux hosts, and is not supported on Docker Desktop for Mac, Docker Desktop for Windows, or Docker EE for Windows Server.

Let us verify this.

$ docker container run -d --network host httpd

You will see that apache server is running on host port 80 without even publishing this port. This is because the container is using host network itself.

## **Adding and removing containers to network dynamically**

Container can be attached and removed from any network dynamically.

Let’s see how to do this.

1. Run an apache container and attach it to app network

$ docker container run -d --network network1 network2

Create a new network with the name <network2>

$ docker network create network2

 Verify that the network was created

docker network ls

Attach apache container to web network

$ docker network connect network2 <container\_id>

Verify the container is attached to the both networks

$ docker container inspect <container\_id>

Let’s remove the container from network1 network now

$ docker network disconnect network1 <container\_id>

Let’s verify that it was removed from app network

$ docker container inspect <container\_id>

## **Removing existing networks**

As a final step let’s remove all containers and delete all the networks we created.

1. Remove all containers at once

$ docker container rm -f $(docker container ls -aq)

Remove all networks we created

$ docker network rm network1 network2

Overlay Network:

Overlay networks are meant to network containers hosted on different hosts. To create such a network we use the ‘overlay’ driver. The overlay driver is a native driver that helps to create a single layer2 broadcast domain across containers hosted on multiple Docker hosts.