Docker Networking

docker network ls

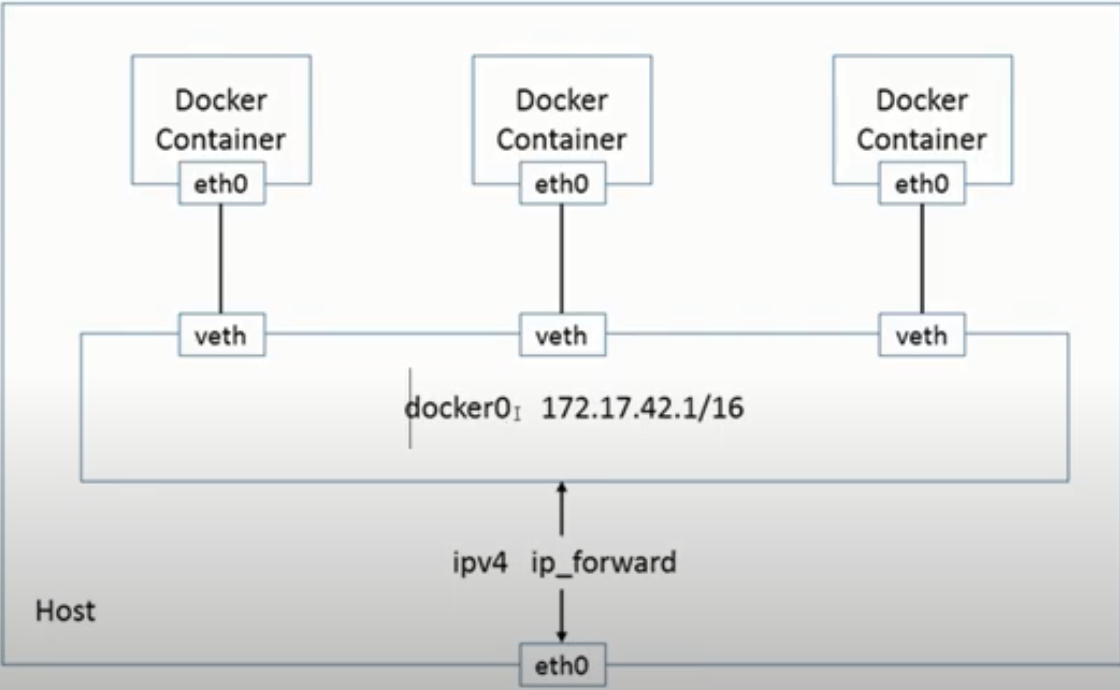
List all network. By default we have some network.

By default, a container is connected to bridge network.

docker network inspect <network name>

Inspect the network. We can see what all containers are attached to this network.

Networking diagram of docker



Run ifconfig command to show all networks and virtual host.

Whenever we attach container to network, it always create pair. One veth will be in docker and another will be inside container.

docker0 - attached an interface.

enp0s3 – nick Interface of host machine.

lo - Loopback

veth

docker container run –P nginx

If we run this command, docker will automatically assign Port 80 of nginx container to the any random port of host machine. Now using that assigned port, we can access nginx container.

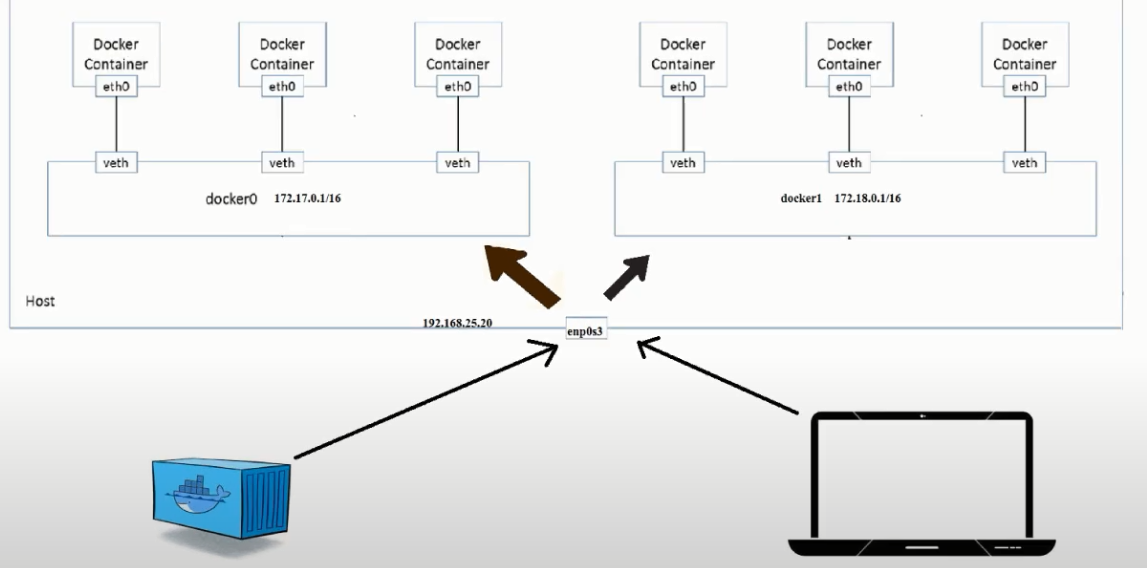
If we want to assign a specific port of host, then we will specify port like below:

Docker container run –p 8080:80 nginx

Docker network create –d <driver name> <network name>

Docker network create –d bridge pankaj

This will create a network named as pankaj and it driver type will be bridge.



Docker container run –it –-network=<network name> <image name> <commands>

Docker container run –it –-network=pankaj ubuntu bash

This command will create container and will attach my custom created network.

Create 2 container and attach them with pankaj network. And create another container attach it with default network.

Now type command as “ip a”. You will see the virtual ethernet and with which container they are connected.

If my two container are attached in same network, then we can ping each other by going inside them directly without opening the port. But if containers are not connected like one is connected to default container and another is connected to pankaj container, then we can’t ping each other, if you want to ping then you have to open the port.

Docker Networking (DNS Enable)

Till now we were pinging using IP address of container. Now we want to ping through hostname.

When we create a custom network, then DNS is by default enabled in that network. Means we can ping a container by it’s hostname from another container, but they should be in same custom network.

Docker provide bridge network by default, in this default bridge network DNS is not enabled. But I want that DNS should be enabled in the bridge network, in that case first you delete the default bridge network and create a custom network with bridge name and bridge driver. Now this bridge network became custom network, and hostname is enabled here also.

# Docker Networking (Host Network)

When we create a container, by default it create a lot of Namespace (PID, NAME, Network, MOUNT, IPC, USERNAME). If we use Host network while creating container, then our NETWORK namespace will not be isolated. My entire container will be isolated, except NETWORK namespace.

If you will create container using Host network, then type “ifconfig” command in docker host and docker container, you will find same output.

We can’t create more than one host network. By default, docker network give us a host name network whose driver type is host. If we try to create custom network with another name and give it’s driver type as host, then it will show error.

# Docker Networking (Null Network, None Network)

If you want that you container should not attach to any network, then assign None network to that container while creating container.

--network=none

# Docker Networking (Connect, Disconnect)

If you want to connect a container with multiple network, then we will use Connect command.

Docker network connect <network name> <container name/id>

Docker network connect pankaj my\_container

Now Create one custom bridge network. Create container with default bridge network. Came out from running container and execute Connect command to connect another custom created network to the running container.

If your container has multiple network, and you want to disconnect any network, then use Disconnect command.

Docker network disconnect <network name> <container name>

Docker network disconnect pankaj my\_container

Question 1:- Suppose I created a container with None network, now can I assign my custom bridge network to that container.

Question 1:- Suppose I created a container with host network, now can I assign my custom bridge network or None network to that container.

# Docker Networking (Remove, Proun)

Docker network rm <network name>

Docker network rm pankaj

* Will remove container

Docker network prune

* Will remove all unused network.

# Docker Compose: Overview

First, we have to install docker compose. Docker-compose file can be in .yml or .json format.

docker-compose up

* This command will search docker-compose.yml file in current directory and will create container, volume, network which are define in docker-compose.yml file.

docker-compose down

* This command will stop all container/network which are define in docker-compose.yml file. This command will not delete Volumes.

docker-compose down –-volume

* This command will delete everything including Volumes.

If suppose you are creating 2 container of nginx and doing port mapping on host port as 8000:80, 8001:80. Now after running docker-compose up command, 2 container will be created.

Now you are stopping these container using docker-compose down command.

Again you make changes in same docker file and you change only in the last container’s Port like 8002:80, in that case the first container will be taken from stopped container and 2nd container will be re-created.

If suppose you are not giving docker-compose.yml name to the file, then you have to explicitly specify the file name like below:

Docker-compose –f <yaml file name> up

Docker-compose –f pankaj.yml up

Docker-compose create

* This will only create container from docker-compose file, it will not start them. But this command will not create network.

Docker-compose rm

* This will remove the earlier created container.

Docker-compose up –-no-start

* This command work similar to Create command. But this command will create Network also.

Docker-compose start

* This command will start the container which are defined in docker-compose file. But make sure that you have earlier created these container using Create command.

Docker-compose stop

* This command will stop the running container which are defined in docker-compose file.

Docker-compose rm

* This command will remove all containers which are defined in the docker-compose file. Make sure this command will not delete the Network, which is defined in docker-compose fie.

Docker-compose images

* This will not give any output. Because docker-compose never build docker image.

Docker-compose pause

* This command will pause the Container, which are in docker-compose.yml file.

Docker-compose unpause

* This command will unpause the Container, which are in docker-compose.yml file.

Docker-compose ps

* This command will tell the status of all the container, which are in docker-compose file.

Docker-compose kill

* This command will kill the containers, which are in docker-compose file

Docker-compose port <service\_name> <port\_number>

Docker-compose port my\_nginx 80

* This command will tell that which host port is mapped to my port 80 of my\_nginx.

Docker-compose logs –f

* This command will attached to the terminal. Now whatever container you will access, it will show you all the current logs.

Docker-compose exec <service name> <command>

Docker-compose exec pankaj ls

* This command will execute specified command in the specified running container. And will show the output in your current screen.

Docker-compose run <service name> <command>

Docker-compose run pankaj ls

* This command will create a new container from pankaj service and execute specified command in that newly created container and finally exit from that container and display the output in your current screen.

Docker-compose restart

* This command will restart the container

Docker-container pull

* This command will only pull the image from docker hub.

Docker-container scale <service name=no of new container> <service name=no of new container>

Docker-container scale pankaj=4 gupta=1

* This command will create new container of the specified service as the number specified. We can specify as many service and as many numbers.

Docker-compose top

* This command will tell the running process inside running container.

## Compose File Syntax

A docker-compose.yml file is organized into five sections:

version: ‘<version number’

version: ‘3’

* Specifies the Compose file syntax version. This guide will use Version 3 throughout.

Services

* In Docker a service is the name for a “Container in production”. This section defines the containers that will be started as a part of the Docker Compose instance.

Networks

* This section is used to configure networking for your application. You can change the settings of the default network, connect to an external network, or define app-specific networks.

Volumes

* Mounts a linked path on the host machine that can be used by the container.

Configs

* Specify config details

Most of this guide will focus on setting up containers using the services section. Here are some of the common directives used to set up and configure containers:

image: <image name>

image: ubuntu

* Sets the image that will be used to build the container.

Build

build: ./dir

* This directive can be used instead of image. Specifies the location of the Dockerfile that will be used to build this container. This means that Dockerfile is at ./dir location.
* If we specify build and image both, then Compose names the built image with the specified image name.
* Build option is ignored when deploying a stack in swarm mode.

restart: <action>

restart: always

* Tells the container to restart if the system restarts or after a crash.

Volumes

* Mounts a linked path on the host machine that can be used by the container

Environment

* Define environment variables to be passed in to the Docker run command.

depends\_on

* Sets a service as a dependency for the current block-defined container

Port

* Maps a port from the container to the host in the following manner: host:container
* Exm:-
* ports:

-“host port:container port”

Links

* Link this service to any other services in the Docker Compose file by specifying their names here.

container\_name: <container custom name>

container\_name: my\_container

* The container\_name directive is used to override the randomly generated container name and replace it with a name that is easier to remember and work with.

**Build**

* Configuration options that are applied at build time.
* build can be specified either as a string containing a path to the build context:
  + version: "3.8"
  + services:
    - webapp:
      * build: ./dir
* Or, as an object with the path specified under context and optionally Dockerfile and args:
  + version: “3.8”
  + services:
    - webapp:
      * build:
        + context : ./dir
        + dockerfile: Dockerfile-alternate
        + args:

buildno: 1

* If you specify image as well as build, the Compose names the build image with the webapp and optional tag specified in image:
  + build: ./dir
  + image: webapp:tag
  + This results in an image named webapp and tagged tag, build from ./dir.
* The build option is ignored when deploying a stack in swarm mode. The docker stack command does not build images before deploying.

**Context**

* Either a path to a directory containing a Dockerfile, or a url to a git repository.
* When the value supplied is a relative path, it is interpreted as relative to the location of the Compose file. This directory is also the build context that is sent to the Docker daemon.
  + build
    - context: ./dir

**Dockerfile**

* Alternate Dockerfile.
* Compose uses an alternate file to build with. A build path must also be specified.
  + Build:
    - Context: ./dir
    - Dockerfile: Dockerfile

**Args**

* Add build arguments, which are environment variables accessible only during the build process.
* First, specify the arguments in your Dockerfile:
  + ARG buildno
  + ARG gitcommithash
  + RUN echo “Build number: $buildno”
  + RUN echo “Git commit hash: $gitcommithash”
* Then specify the arguments under the build key. You can pass a mapping or a list:
  + build:
    - context: ./dir
    - args:
      * buildno: 1
      * gitcommithash: abcd
  + build
    - context: ./dir
    - args:
      * - buildno=1
      * - gitcommithash=abcd
* In your Dockerfile, if you specify ARG before the FROM instruction, ARG is not available in the build instructions under FROM. If you need an argument to be available in both places, also specify it under the FROM instruction.
* You can omit the value when specifying a build argument, in which case its value at build time is the value in the environment where Compose is running.
* YAML boolean values ("true", "false", "yes", "no", "on", "off") must be enclosed in quotes, so that the parser interprets them as strings.

**cache\_from**

* A list of images that the engine uses for cache resolution.
  + build:
    - context: ./dir
    - cache\_from:
      * - alpine:latest
      * - core/web\_app:3.14

**labels**

* Add metadata to the resulting image using Docker labels. You can use either an array or a dictionary.
* It’s recommended that you use reverse-DNS notation to prevent your labels from conflicting with those used by other software.
  + build:
    - context: ./dir
    - labels:
      * com.example.department: “Finance”
  + build:
    - Context: ./dir
    - Labels:
      * - “com.example.department=Finance”

**network**

* Set the network containers connect to for the RUN instructions during build.
  + build:
    - context: ./dir
    - network: host
* Use **none** to disable networking during build:
  + build:
    - context: ./dir
    - network: none

**target**

* Build the specified stage as defined inside the Dockerfile.
  + build:
    - context:
    - target: prod

**cap\_add, cap\_drop**

* Add or drop container capabilities. See man 7 capabilities for a full list.
  + Cap\_add:
    - - ALL
  + Cap\_drop:
    - - NET\_ADMIN
    - - SYS\_ADMIN
* The cap\_add and cap\_drop options are ignored when deploying a stack in swarm mode.

**Command**

* Override the default command.
  + command: bundle exec thin -p 3000
* The command can also be a list, in a manner similar to dockerfile:
  + command: ["bundle", "exec", "thin", "-p", "3000"]

**config**

* Grant access to configs on a per-service basis using the per-service configs configuration. The config must already exist or be defined in the top-level configs configuration of this stack file, or stack deployment fails.
* Two different syntax variants are supported.
* **SHORT SYNTAX**
  + The short syntax variant only specifies the config name. This grants the container access to the config and mounts it at /<config\_name> within the container. The source name and destination mountpoint are both set to the config name.
  + The following example uses the short syntax to grant the redis service access to the my\_config and my\_other\_config configs. **The value of my\_config is set to the contents of the file ./my\_config.txt, and my\_other\_config is defined as an external resource**, which means that it has already been defined in Docker, either by running the docker config create command or by another stack deployment. If the external config does not exist, the stack deployment fails with a config not found error.
  + version: "3.8"
  + services:
    - redis:
      * image: redis:latest
      * deploy:
        + replicas: 1
      * configs:
        + - my\_config
        + - my\_other\_config
  + configs:
    - my\_config:
      * file: ./my\_config.txt
    - my\_other\_config:
      * external: true
* **LONG SYNTAX**
  + The long syntax provides more granularity in how the config is created within the service’s task containers.
  + Source:
    - The name of the config as it exists in Docker.
  + Target:
    - The path and name of the file to be mounted in the service’s task containers. Defaults to /<source> if not specified.
  + UID and GID:
    - The numeric UID or GID that owns the mounted config file within in the service’s task containers. Both default to 0 on Linux if not specified. Not supported on Windows.
  + Mode:
    - The permissions for the file that is mounted within the service’s task containers, in octal notation. For instance, 0444 represents world-readable. The default is 0444. Configs cannot be writable because they are mounted in a temporary filesystem, so if you set the writable bit, it is ignored. The executable bit can be set.
  + The following example sets the name of my\_config to redis\_config within the container, sets the mode to 0440 (group-readable) and sets the user and group to 103. The redis service does not have access to the my\_other\_config config.
  + version: "3.8"
  + services:
    - redis:
      * image: redis:latest
      * deploy:
        + replicas: 1
      * configs:
        + - source: my\_config
        + target: /redis\_config
        + uid: '103'
        + gid: '103'
        + mode: 0440
  + configs:
    - my\_config:
      * file: ./my\_config.txt
    - my\_other\_config:
      * external: true
* You can grant a service access to multiple configs and you can mix long and short syntax. Defining a config does not imply granting a service access to it.

**Container\_name**

* Specify a custom container name, rather than a generated default name.
  + Container-name: my\_container
* Because Docker container names must be unique, you cannot scale a service beyond 1 container if you have specified a custom name. Attempting to do so results in an error.
* The container\_name option is ignored when deploying a stack in swarm mode.

**Depends\_on:**

* Express dependency between services.
* ”**docker-compose up**” starts services in dependency order. In the following example, “db” and ”redis” are started before ”web”.
* ”**docker-compose up** **SERVICE**” automatically includes ”SERVICE’s” dependencies. In the example below, ”docker-compose up” web also creates and starts ”db” and ”redis”.
* ”**docker-compose stop**” stops services in dependency order. In the following example, ”web” is stopped before ”db” and ”redis”.
* version: "3.8"
* services:
  + web:
  + build: .
  + depends\_on:
    - - db
    - - redis
  + redis:
    - image: redis
  + db:
    - image: postgres
* ”depends\_on” does not wait for ”db” and ”redis” to be “ready” before starting web - only until they have been started. If you need to wait for a service to be ready, see Controlling startup order for more on this problem and strategies for solving it.
* Version 3 no longer supports the condition form of ”depends\_on”.
* The ”depends\_on” option is ignored when deploying a stack in swarm mode with a version 3 Compose file.

**Deploy:**

* Specify configuration related to the deployment and running of services. This only takes effect when deploying to a swarm with docker stack deploy, and is ignored by docker-compose up and docker-compose run.
* Several sub-options are available:

**endpoint\_mode:**

* Specify a service discovery method for external clients connecting to a swarm.
* endpoint\_mode: vip
  + Docker assigns the service a virtual IP (VIP) that acts as the front end for clients to reach the service on a network. Docker routes requests between the client and available worker nodes for the service, without client knowledge of how many nodes are participating in the service or their IP addresses or ports. (This is the default.)
* endpoint\_mode: dnsrr
  + DNS round-robin (DNSRR) service discovery does not use a single virtual IP. Docker sets up DNS entries for the service such that a DNS query for the service name returns a list of IP addresses, and the client connects directly to one of these. DNS round-robin is useful in cases where you want to use your own load balancer, or for Hybrid Windows and Linux applications.
* services:
  + mysql:
    - image: mysql
    - deploy:
      * mode: replicated
      * replicas: 2
      * endpoint\_mode: dnsrr/vip

**mode**

* Either global (exactly one container per swarm node) or replicated (a specified number of containers). The default is replicated.
* services:
  + worker:
    - image: dockersamples/examplevotingapp\_worker
    - deploy:
      * mode: global

**replicas**

* If the service is replicated (which is the default), specify the number of containers that should be running at any given time.
* services:
  + worker:
    - image: dockersamples/examplevotingapp\_worker
    - deploy:
      * mode: replicated
      * replicas: 6

**max\_replicas\_per\_node**

* If the service is replicated (which is the default), limit the number of replicas that can run on a node at any time.
* When there are more tasks requested than running nodes, an error no suitable node (max replicas per node limit exceed) is raised.
* services:
  + worker:
    - image: dockersamples/examplevotingapp\_worker
    - deploy:
      * mode:replicated
      * replicas: 6
      * placement:
        + max\_replicas\_per\_node: 1

**labels**

* Specify labels for the service. These labels are only set on the service, and not on any containers for the service.
* services:
  + webapp:
    - image: web
    - deploy:
      * labels:
        + description: “This is description”
* To set labels on containers instead, use the labels key outside of deploy:
* services:
  + webapp:
    - image: web
    - labels:
      * description: “This is description”

**restart\_policy**

* Configures if and how to restart containers when they exit. Replaces restart.
* Condition
  + One of none, on-failure or any (default: any).
* Delay
  + How long to wait between restart attempts, specified as a duration (default: 0).
* max\_attempts
  + How many times to attempt to restart a container before giving up (default: never give up). If the restart does not succeed within the configured window, this attempt doesn’t count toward the configured max\_attempts value. For example, if max\_attempts is set to ‘2’, and the restart fails on the first attempt, more than two restarts may be attempted.
* Window
  + How long to wait before deciding if a restart has succeeded, specified as a duration (default: decide immediately).
* services:
  + redis:
    - image: redis:alpine
    - deploy:
      * restart\_policy:
        + condition: on-failure
        + delay: 5s
        + max\_attmepts: 3
        + window: 120s

**entrypoint:**

* Override the default entrypoint.
* The entrypoint can also be a list, in a manner similar to dockerfile:
  + entrypoint: /code/entrypoint.sh
  + entrypoint: ["php", "-d", "memory\_limit=-1", "vendor/bin/phpunit"]
* Setting entrypoint both overrides any default entrypoint set on the service’s image with the ENTRYPOINT Dockerfile instruction, and clears out any default command on the image - meaning that if there’s a CMD instruction in the Dockerfile, it is ignored.

**environment:**

* Add environment variables. You can use either an array or a dictionary. Any boolean values (true, false, yes, no) need to be enclosed in quotes to ensure they are not converted to True or False by the YML parser.
* Environment variables with only a key are resolved to their values on the machine Compose is running on, which can be helpful for secret or host-specific values.
* environment:
  + RACK\_ENV: development
  + SHOW: ‘true’
  + SESSION\_SECRET:
* environment:
  + - RACK\_ENV= development
  + - SHOW=true
  + – SESSION\_SECRET
* If your service specifies a build option, variables defined in environment are not automatically visible during the build. Use the args sub-option of build to define build-time environment variables.

**expose:**

* Expose ports without publishing them to the host machine - they’ll only be accessible to linked services. Only the internal port can be specified.
* expose:
  + - “3000”
  + - “8000”