Docker - Advanced

2026: cyhuang
2025: phkoan

Outline

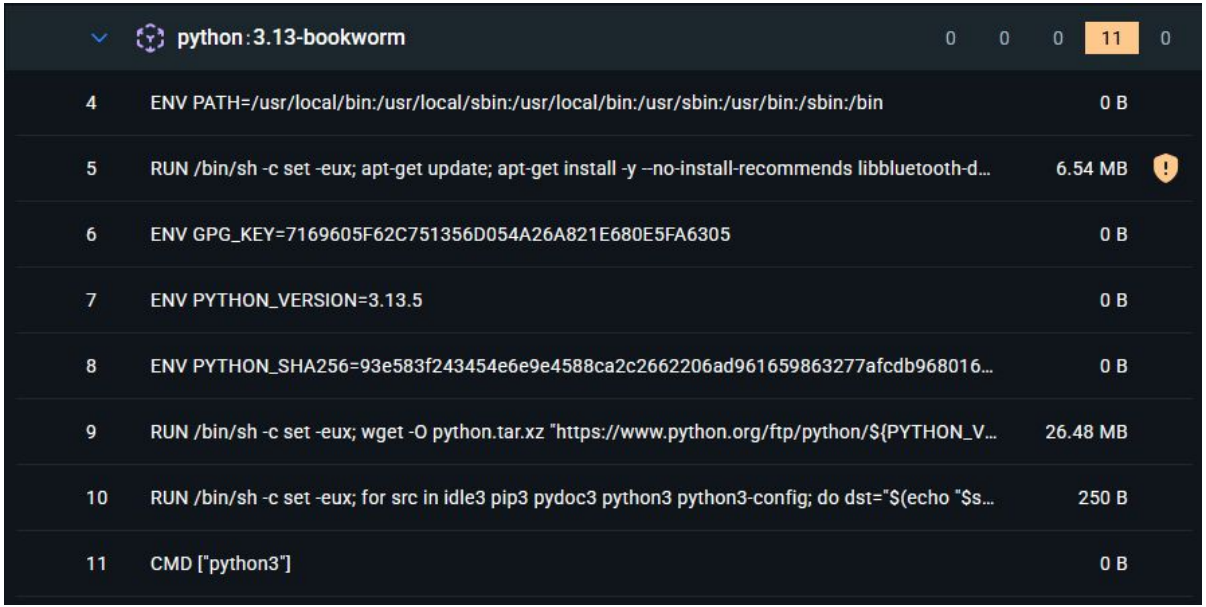
- Dockerfile & docker build
- Docker Network
- Docker Compose

Dockerfile

Dockerfile

- Most of the time, the images built by others cannot satisfy our needs.
  - We may want additional packages, specific system configuration, etc.
- Dockerfile allows users to build their own container image.

Recap: Container Image Layers



	python:3.13-bookworm	0	0	0	11	0
4	ENV PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin				0 B	
5	RUN /bin/sh -c set -eux; apt-get update; apt-get install -y --no-install-recommends libbluetooth-d...				6.54 MB	
6	ENV GPG_KEY=7169605F62C751356D054A26A821E680E5FA6305				0 B	
7	ENV PYTHON_VERSION=3.13.5				0 B	
8	ENV PYTHON_SHA256=93e583f243454e6e9e4588ca2c2662206ad961659863277afcdb968016...				0 B	
9	RUN /bin/sh -c set -eux; wget -O python.tar.xz "https://www.python.org/ftp/python/${PYTHON_V...				26.48 MB	
10	RUN /bin/sh -c set -eux; for src in idle3 pip3 pydoc3 python3 python3-config; do dst="$(echo "$s...				250 B	
11	CMD ["python3"]				0 B	

source: python:3.13:bookworm

Dockerfile

- Dockerfile consists of multiple lines of instructions.
- Common instructions
  - `FROM <image>`: The base image
  - `RUN <command>`: The command to execute in container
  - `COPY <host path> <container path>`: Copy file or dir from host to container
  - `WORKDIR <path>`: The working directory path (pwd) in container
  - `CMD [<command>...]`: Default startup commands
  - `ENTRYPOINT [<command>...]`: The main command of container
  - `ENV <key>=<value>`: The environmental variable in container

Dockerfile

- We can build a image with docker build.
- Usage: `docker build <Dockerfile directory>`
- Frequently used options
  - `-t <name>[:<tag>]`: The image name and tag

.dockerignore

- The `COPY . .` instruction can copy unnecessary files to container such as node modules, build cache. etc.
- The file `.dockerignore` can prevent you from copying these files to container and reduce the size of image.

Dockerfile - Example

```
FROM python:3.13-bookworm

RUN apt-get update

RUN apt-get install -y curl vim git build-essential net-tools

COPY . .

RUN pip install --no-cache-dir -r requirements.txt

CMD ["python", "main.py"]
```

CMD vs. ENTRYPOINT

- `CMD` and `ENTRYPOINT` both defines what to do when a container starts.
- `ENTRYPOINT` cannot be overriden by `docker run <command>`.
  - It's more like forcing you to run.
  - But it can be overriden by `docker run --entrypoint`.
- `CMD` can be overriden by `docker run <command>` since it provides the "default" commands.
- When `ENTRYPOINT` and `CMD` are both defined, `CMD` become the arguments of `ENTRYPOINT`.
  - For example, `ENTRYPOINT ["echo"] and CMD ["Hello world!"]`.
- We often write `entrypoint.sh` and use `ENTRYPOINT ["./entrypoint.sh"]` in Dockerfile.

Dockerfile - Build Args

- When building an image, we can pass arguments to Dockerfile.
  - Add `ARG MY_VAR` in Dockerfile
  - Run `docker build --build-arg MY_VAR=hello`.
- The arguments can be used in the Dockerfile using the `${MY_VAR}` syntax.

```
ARG MY_VAR

RUN echo ${MY_VAR} > /tmp/my_file
```

Container Image Layer Cache

- When there are nothing changes in Dockerfile or related files, the built layers will be cached.
  - When users re-build, only the changes layers are re-built.
  - For example, if the line of apt install is not changed, the layer will be cached.
  - Another example, if the Python files are not changed, the `COPY` line will be cached.
- However, if a layer is modified, the layers after it require re-build.

Utilize Cache to Save Build Time

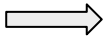
- Let's see the lines of `COPY . .` and `RUN pip install`.
- What will happen if we change our code?
  - `COPY . .` copies all files, including your Python code and requirements.txt.
  - Your code changes, so `COPY . .` needs re-build.
  - All lines after it require re-build.
- Hence, `RUN pip install` will be run again, even though `requirements.txt` is not changed.

```
...
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "main.py"]
```

Utilize Cache to Save Build Time

- `RUN pip install` is time-consuming, so we want to run it only when necessary.
  - Necessary case: `requirements.txt` is changed.
- We can reorder the instructions to avoid unnecessary re-build.
  - Only copy `requirements.txt` before pip install, and copy the Python files after the installation.

```
...
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "main.py"]
```



```
...
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "main.py"]
```

Utilize Cache to Save Build Time

- This technique applies to many cases.
  - In Node.js, only copy `package.json` before `npm install`.
  - In C++, only copy `conanfile.txt` before `conan install`.
- Principle: Place time-consuming Dockerfile instructions before frequently changing instructions.
  - Obviously, we will not place `apt install` before copying the code.

Reduce Image Size - TL;DR

- Do not copy unnecessary files.
  - Exclude them by using `.dockerignore`.
- Reduce the number of layers.
- Use a slim or Alpine-based image.
- Use multiple stages building.

Reduce Image Size - Reduction of Layers

- The size of image is positively correlated with the number of layers.
- We can reduce the number of RUN instructions by combining them together.
  - Note: This may increase the likelihood of cache invalidation.

```
RUN apt update
RUN apt install -y vim tmux git
RUN apt install -y python3
...
```



```
RUN apt update && \
    apt install -y vim tmux git python3
...
```

Reduce Image Size - Slim Image

- Take the image of Python as example
  - `3.13.5-bookworm`: 366.1 MB
  - `3.13.5-slim-bookworm`: 43.43 MB
  - `3.13.5-alpine3.22`: 16.07 MB
- Use slim image can greatly reduce the size of the image.
- However, the slim images trim some packages, so there may be compatibility issues.
  - In Debian slim images, iproute2 package is not installed.
  - Alpine-based image uses MUSL libs, instead of common glibc.

Multiple Stages Building

- Consider a scenario:
  - You have a C++ project, and you want to use Docker to build and run it.
  - You install some system packages and some C++ libraries in the image.
  - You run `make` and build your application.
  - You run this program in your container, and it works.

```
RUN apt update && apt install -y libssl-dev python3 python3-pip cmake make gcc && \

    pip install conan --break-system-packages

COPY conanfile.txt .

RUN conan install .

COPY . .

RUN cmake -S . -B build && make -C build -j8

CMD ["./build/bin/my_app"]
```

Multiple Stages Building

- You run the application, which is a binary, in a 1GB container.
  - Do we really need all the things when running the application? No!
- All we need is merely glibc and the SSL library.

```
RUN apt update && apt install -y libssl-dev

CMD ["./build/bin/my_app"]
```

Multiple Stages Building

```
FROM debian:trixie-slim AS builder

WORKDIR /app

RUN apt update && \

    apt install -y libssl-dev python3 python3-pip \

                   cmake make gcc && \

    pip install conan --break-system-packages

COPY conanfile.txt .

RUN conan install .

COPY . .

RUN cmake -S . -B build && make -C build -j8
```

```
FROM debian:trixie-slim

WORKDIR /app

RUN apt update && apt install -y libssl-dev

COPY --from=builder /app/build/bin/my_app ./my_app

CMD ["./my_app"]
```

Docker Network

Docker Network Drivers

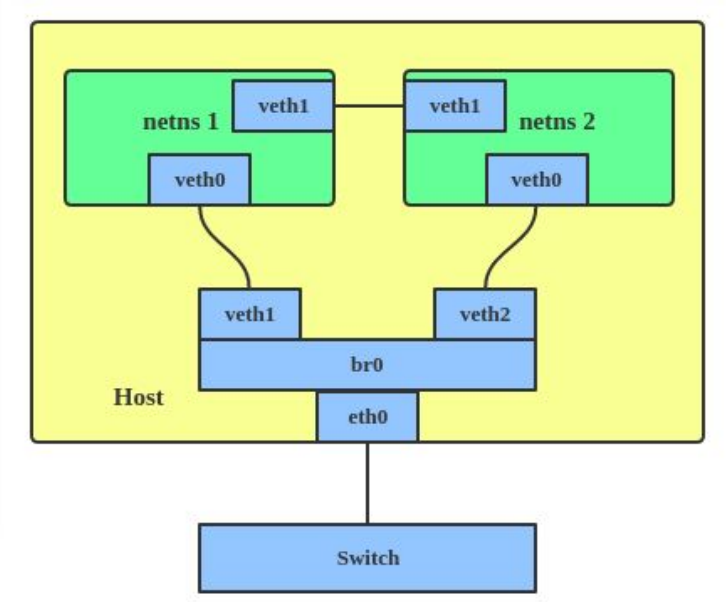
- Docker provides multiple network drivers for difference use cases.
- We will focus on bridge (default) and host.

Recap: veth

- A special type of virtual interface and always comes in pair.
- Two veth interfaces can be in different namespace.
  - It enables cross-namespace communication.
- Veth transmits packets from/to peer veth
  - You can think the link between veth pair as an ethernet cable.

Recap: Bridge

- A special type of virtual interfaces
- Acting like a light-weight virtual switch or hub
- Physical/Virtual network interface can connected to it.
- Bridge can forward packet to/from connected interface.

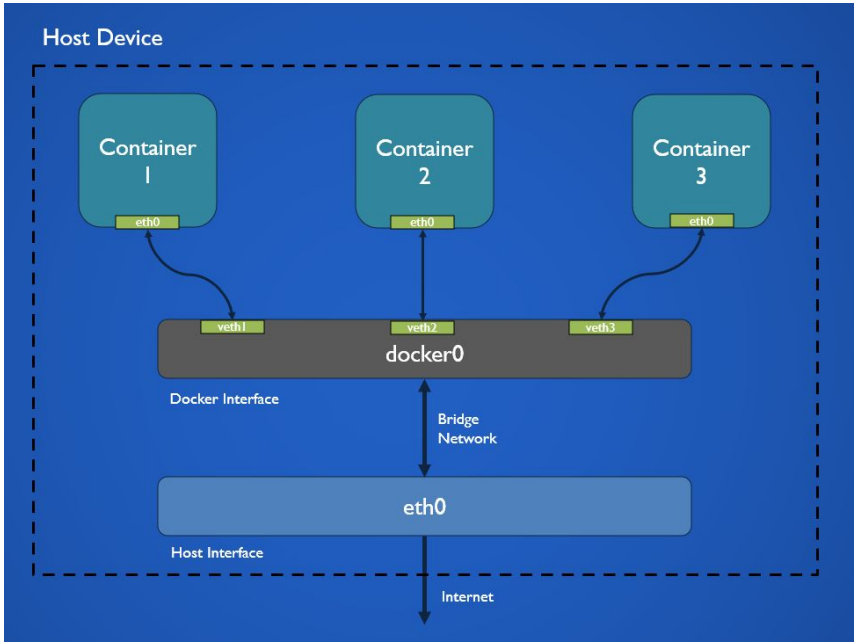


source: [Introduction to Linux interfaces for virtual networking | Red Hat]

Docker Network Drivers - Bridge

- Bridge is the default driver.
- It uses Linux bridge and veth to connect Linux network namespaces to eth0.

Source: Docker 101: Part 4 - Networks | Source Code



Docker Network Drivers - Bridge

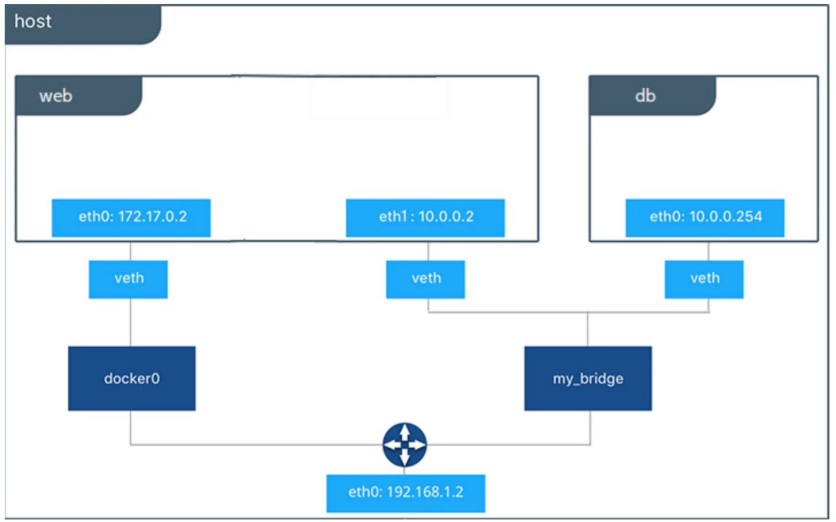
The above one is in container, while the below one is in host.

```
2: eth0@if69: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether ce:ab:5d:13:e9:3f brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

```
9: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 0e:b2:95:d4:e0:0b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
69: veth7308954@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether ca:c5:86:04:4f:4c brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Docker Network Drivers - Bridge

- If we create containers without additional network configuration, the container will
  - Use bridge network driver.
  - Attach the default bridge: `docker0`.
- The containers can communicate with each other.
  - What if we do not want so?



source: [Introduction to Container Networking in Docker | taikun.cloud]

Docker Network Drivers - Bridge

- We can use `docker network create --driver bridge my_bridge` to create a new bridge interface.
  - We can see the new bridge via `ip link` and `docker network ls`.
- And use `docker run --network my_bridge debian bash` to create a container attached to the new bridge.

Docker Network Drivers - Bridge

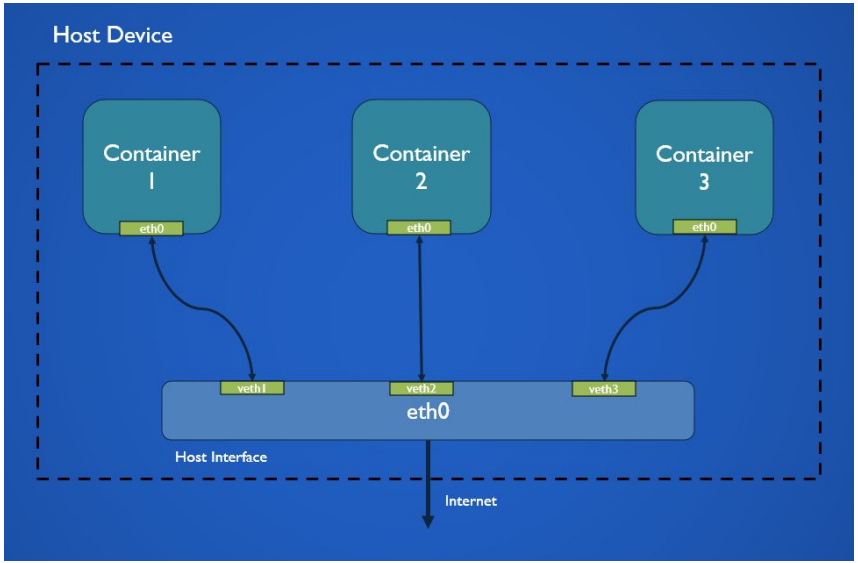
```
NETWORK ID     NAME         DRIVER    SCOPE
30ed72de30c7   my_bridge    bridge    local
295b421f0b83   bridge       bridge    local
```

```
9: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 0e:b2:95:d4:e0:0b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
72: br-30ed72de30c7: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether ba:1c:bd:a0:8b:c6 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global br-30ed72de30c7
       valid_lft forever preferred_lft forever
```

Docker Network Drivers - Host

- Host driver allows containers to directly bind to `eth0`.
- No network stack isolation.
  - A container sees the same network stack as host.
  - A port used by the host cannot be used by a container at the same time, and vice versa.

Source: Docker 101: Part 4 - Networks | Source Code



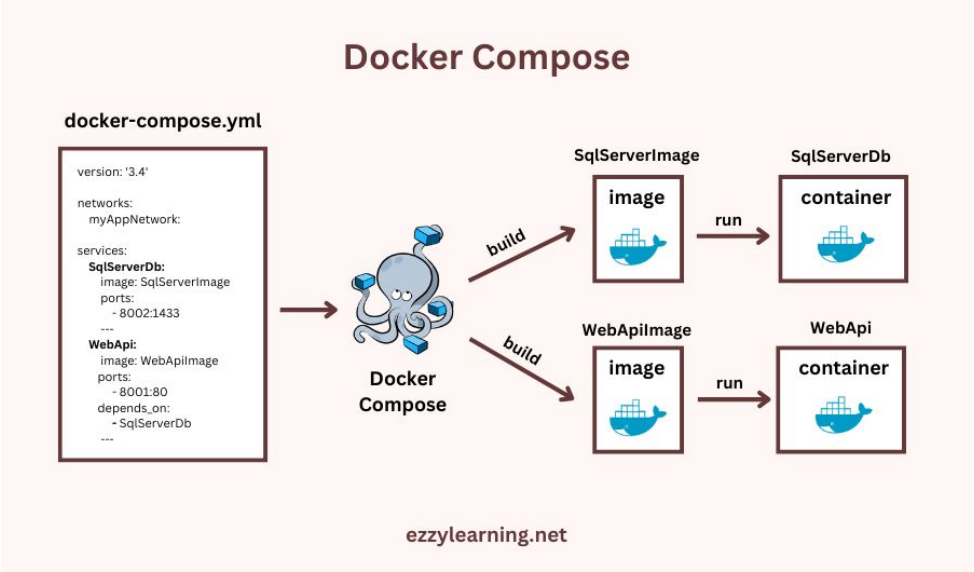
FYR - Docker Network Drivers - Others

- `none`
  - Creates a network namespace containing only the loopback interface.
- `macvlan`
  - Allows containers to appear as physical devices on the network by assigning them their own unique MAC addresses.
- `ipvlan`
  - Allows containers to share a single MAC address while having their own IP addresses.

Docker Compose

Docker Compose

- Docker Compose lets you define and manage multi-container Docker applications using a simple YAML configuration file.



source: [Dockerize ASP.NET Core API and SQL Server]

Docker Compose - Config File Example

```
services:
  web:
    build: ./app
    ports:
      - "5000:5000"
    environment:
      - DB_URL=postgresql://postgres:postgres@db:5432/my_app
    depends_on:
      - db
    volumes:
      - ./app:/app
```

```
  db:
    image: postgres:16.9-bookworm
    restart: always
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: mydatabase
    volumes:
      - ./pgdata:/var/lib/postgresql/data
```

Docker Compose - Commands

- With `docker-compose.yml`, we can start to use Docker Compose.
- Frequently used commands
  - `docker compose up -d`: Start or re-build (when YAML is changed) the containers.
  - `docker compose down`: Remove all containers and networks.
  - `docker compose start/restart/stop [<container name>]`: Start/Restart/Stop the containers.
  - Most of the commands in Docker have corresponding version in Docker Compose.
    - For example, `docker compose logs`.
- Frequently used options
  - `-f`: Specify the location of `docker-compose.yml`.

Docker Compose - Image

```
build:
  context: backend
  args:
    - MY_VAR=hello
```

- `image`: The image to launch the container
- `build`
  - `context`: The Docker build context
    - In most case, it is the directory containing Dockerfile.
    - If you want to `COPY ../common /app/common` in your backend Dockerfile, you need to specify `context` as the parent of backend dir and `dockerfile` as `backend/Dockerfile`, or `COPY` will fail.
  - `dockerfile`: The location of Dockerfile (not the directory)
  - `args`: The build args
- If `image` and `build` both are specified, the image indicates the name of the built image.

Docker Compose - Dependency

- Docker Compose allows container dependency by using the `depends_on` setting.
- `depends_on` containing only the service names make the dependency start first.
  - No checks for container failure.
- We can specify condition for advanced checks.
  - `service_started`: Same as the above one.
  - `service_completed_successfully`: The exit code must be 0.
  - `service_healthy`: The health check must pass.

```
depends_on:
  - db
```

```
depends_on:
  db:
    condition: service_healthy
```

Docker Compose - Restart Policy

- Docker provides automatic restart feature.
- The modes
  - `no`: No restart. The default option.
  - `on-failure`: Restart the container if exit code is not 0.
  - `always`: Restart the container once it's stopped (except manual stop).
  - `unless-stopped`: Restart the container once it's stopped (except manual stop).
- `always` vs. `unless-stopped`: `always` will restart the manually-stopped container after Docker daemon restarts, while `unless-stopped` keeps it stopped.

Docker Compose - Restart Policy

- In Docker Compose, restart policy can be specified with the `restart` keyword.
- In docker run, use `docker run --restart <mode>`.

```
db:
  image: postgres:16.9-bookworm
  restart: always
```

Docker Compose - Health Check

- Docker enables user to run health check for container.
- Two modes:
  - `CMD`: Simply exec.
  - `CMD-SHELL`: Execute command in shell, so shell features such as && are available.
  - In most of cases, `CMD` is recommended.

Docker Compose - Health Check

- Options:
  - `test`: The command to execute.
  - `interval`: The health check interval.
  - `timeout`: The timeout of check.
  - `retries`: The number of allowed failures.
  - `start_period`: The time for container boostrapping (failure is allowed).
  - `start_interval`: The health check interval during start period.

```
healthcheck:
  test: ["CMD", "pg_isready", "-U", "user"]
  interval: 10s
  timeout: 5s
```

Docker Compose - Health Check

This can also be used in Dockerfile and docker run.

```
HEALTHCHECK --interval=10s --timeout=5s \
  CMD pg_isready -U user
```

```
$ docker run \
    --health-cmd="pg_isready -U user" \
    --health-interval=15s \
    --health-timeout=5s \
    --health-retries=3
```

Docker Compose - Network

- By default, all containers use a new bridge network.
  - Generally, this requires no further configuration.
- They can communicate with each other by IP addresses or by domain name (the service name).
- You can configure the network by adding a network section in YAML.
  - Note: The route to configured subnet will be taken by Docker.

```
services:
  coredns:
    image: coredns/coredns
    networks:
      coredns:
        ipv4_address: 192.168.0.2


networks:
  coredns:
    driver: bridge
    ipam:
      config:
        - subnet: 192.168.0.0/16
          gateway: 192.168.0.1
```

FYI: Remove Unused Resources

- By default, the Docker resources will not be removed even though no containers use them.
- You can use `docker <resource> prune` to remove them.
  - Or use `docker system prune` to remove all of them.
- For example,
  - `docker volume prune`
  - `docker network prune`
  - `docker image prune`
  - `docker container prune`
- You can add -a flag to remove "more unused" resources.
  - For example, `docker image prune` removes dangling images, while `-a` removed images without at least one container associated to them.