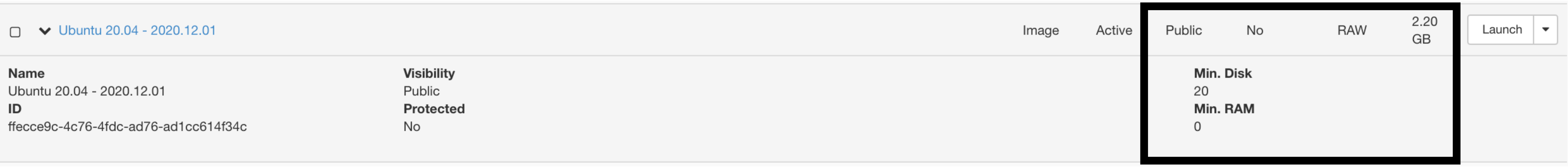
UPPSALA UNIVERSITET

Contextualization and Containers

Salman Toor
salman.toor@it.uu.se

Images and snapshots

Images



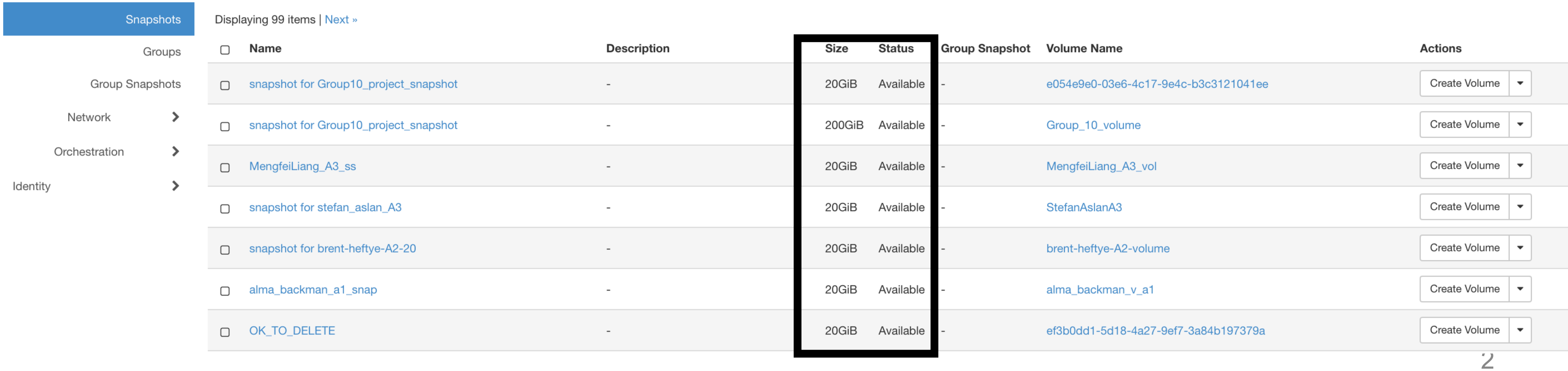
☐	Ubuntu 20.04 - 2020.12.01	Image	Active	Public	No	RAW	2.20 GB	Launch

Name
Ubuntu 20.04 - 2020.12.01
ID
ffecce9c-4c76-4fdc-ad76-ad1cc614f34c

Visibility
Public
Protected
No

Min. Disk
20
Min. RAM
0

Snapshots



Displaying 99 items | Next »

☐	Name	Description	Size	Status	Group Snapshot	Volume Name	Actions
☐	snapshot for Group10_project_snapshot	-	20GiB	Available	-	e054e9e0-03e6-4c17-9e4c-b3c3121041ee	Create Volume
☐	snapshot for Group10_project_snapshot	-	200GiB	Available	-	Group_10_volume	Create Volume
☐	MengfeiLiang_A3_ss	-	20GiB	Available	-	MengfeiLiang_A3_vol	Create Volume
☐	snapshot for stefan_aslan_A3	-	20GiB	Available	-	StefanAslanA3	Create Volume
☐	snapshot for brent-heftye-A2-20	-	20GiB	Available	-	brent-heftye-A2-volume	Create Volume
☐	alma_backman_a1_snap	-	20GiB	Available	-	alma_backman_v_a1	Create Volume
☐	OK_TO_DELETE	-	20GiB	Available	-	ef3b0dd1-5d18-4a27-9ef7-3a84b197379a	Create Volume

Images and formats

- Cloud images are customized disks of OSes for private or public clouds

- Different formats are available:
  - raw: An unstructured disk image format (big in size)
  - vhd: VMware, Xen, Microsoft, VirtualBox, and others
  - vdi: Supported by VirtualBox, QEMU emulator.
  - iso: Archive format for the data contents of an optical disc
  - qcow2: Supported by the QEMU emulator that can expand dynamically and supports Copy on Write.
  - …

http://docs.openstack.org/image-guide/content/image-formats.html

Contextualization

In cloud computing, contextualization means providing a customized computing environment

Or

Allows a virtual machine instance to learn about its cloud environment and user requirement (the ‘context’) and configure itself to run correctly

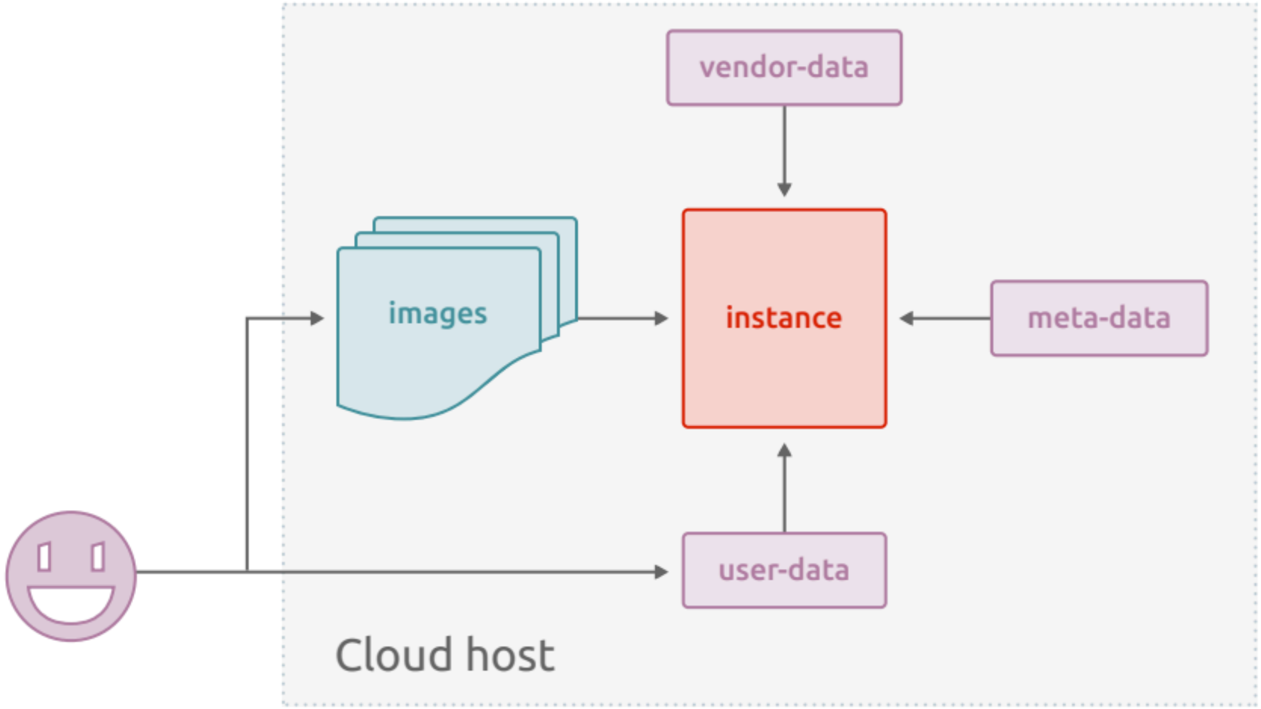
http://www.journalofcloudcomputing.com/content/4/1/17

Contextualization

- Provide scalable solution
- No need to manage fat images
- Dynamic configuration
- Typically work in two layers
  - Meta-data : System information handled at cloud level
  - User-defined-data: User specific requirements/settings

CloudInit

Cloud-init is a standard multi-distribution method for cross-platform cloud instance initialization. It is supported across all major public cloud providers, provisioning systems for private cloud infrastructure, and bare-metal installations.



https://cloudinit.readthedocs.io/en/latest/

https://www.lucd.info/2019/12/06/cloud-init-part-1-the-basics/

CloudInit

```
#cloud-config
apt_update: true
apt_upgrade: true
packages:
 - cowsay
 - python-pip
 - python-dev
 - build-essential
 - cowsay
byobu_default: system

runcmd:
 - echo "export PATH=$PATH:/usr/games" >> /home/ubuntu/.bashrc
 - source /home/ubuntu/.bashrc
 - pip install Flask
 - python /home/ubuntu/cowsay-app.py &
```

Beyond Virtual Machines

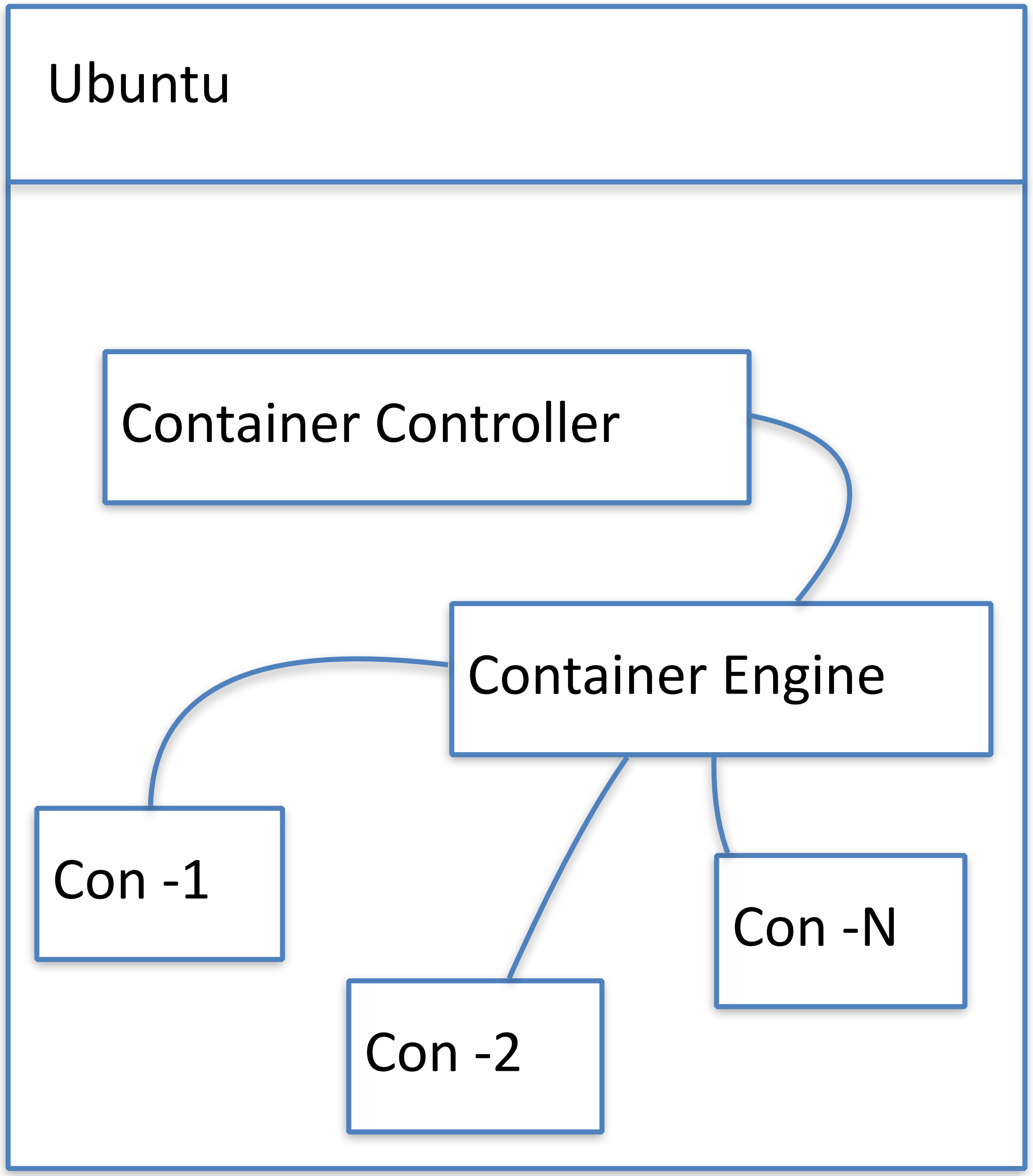
Containers

- OS level virtualization environment
  - Kernelspace is shared
  - Userspace is separate for each linux system (container)
- A lightweight alternative to Virtual Machines (VM)
- Shared same resources as host OS
- A simple model for packaging applications in Linux.

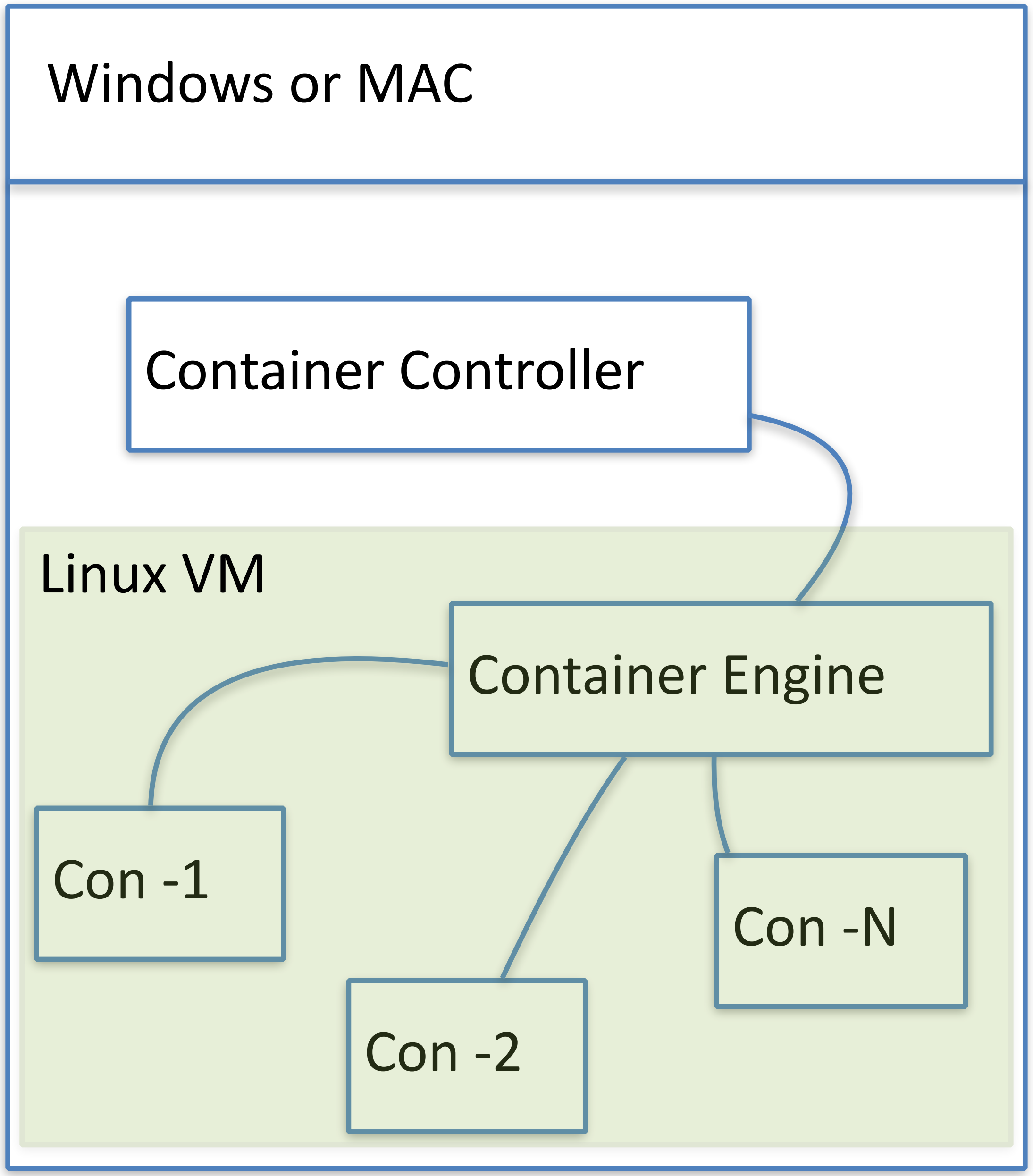
https://linuxcontainers.org/

Basic illustration of containers

Linux based host OS



Non Linux host OS



VM and containers

Its is important to understand that VMs and Containers should not be viewed as competitors

Virtual Machine

- Complete isolation
- Big in size
- High overhead
- Flexible support of multiple OSes
- Greater stability both for hypervisors and VMs
- Better security

Containers

- Application level abstraction
- Lightweight
- Works well with Linux, limited support for Windows
- Weak security
- Significant management Overhead
- Not well suited for large applications
- Important for micro-services design

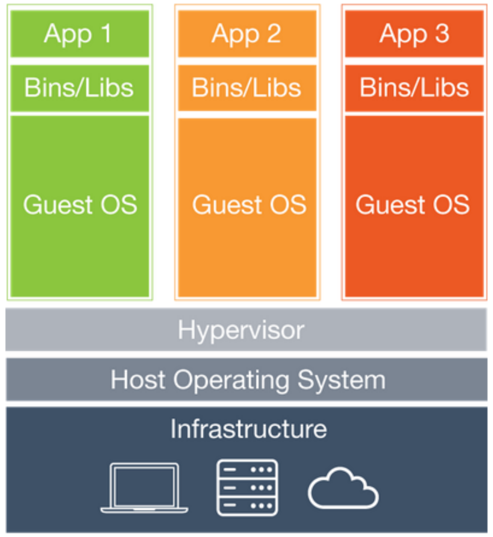
LXC

- LinuX Container (LXC) is an open source software
- Virtual environment based on separate memory, CPU, network, io etc
- Similar to the concept of *chroot*
- Used in most of the container based orchestration tools
- LXD is a newer version of LXC, advanced and stronger support for cloud plugins

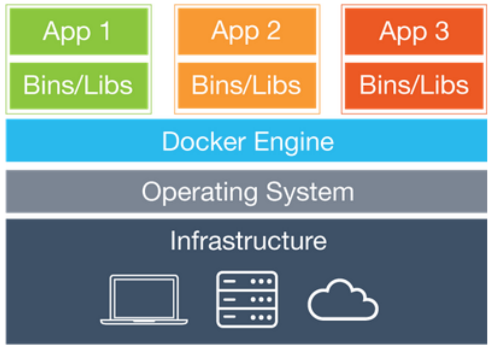
http://blog.scottlowe.org/2013/11/25/a-brief-introduction-to-linux-containers-with-lxc/

Docker

- Docker package an application together with all its dependencies in the container
- Guarantees that it will always run the same regardless of the environment
- Container based orchestration tool
- Docker Hub, container registory
- Open source



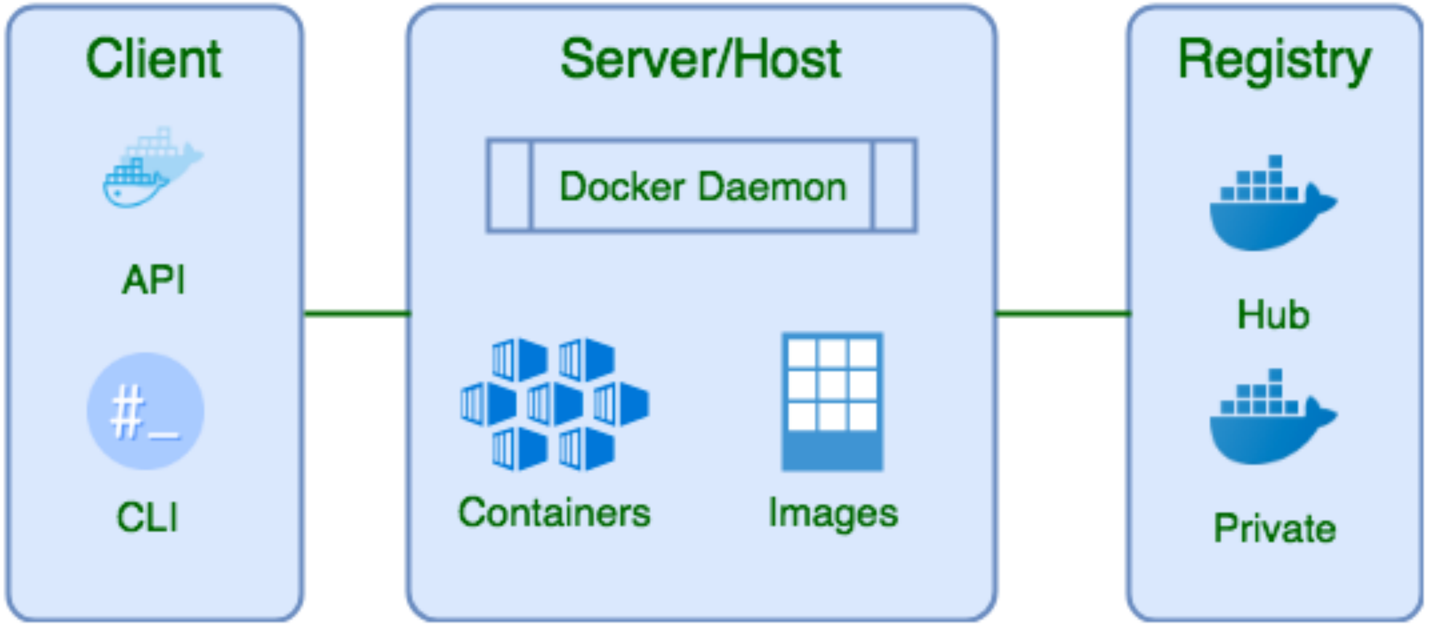
Virtual Machines



Containers

https://www.docker.com/whatisdocker

Docker framework



```
# docker ps
# docker images
# docker build
# docker run
# docker exec
# docker pull
# docker push
….
```

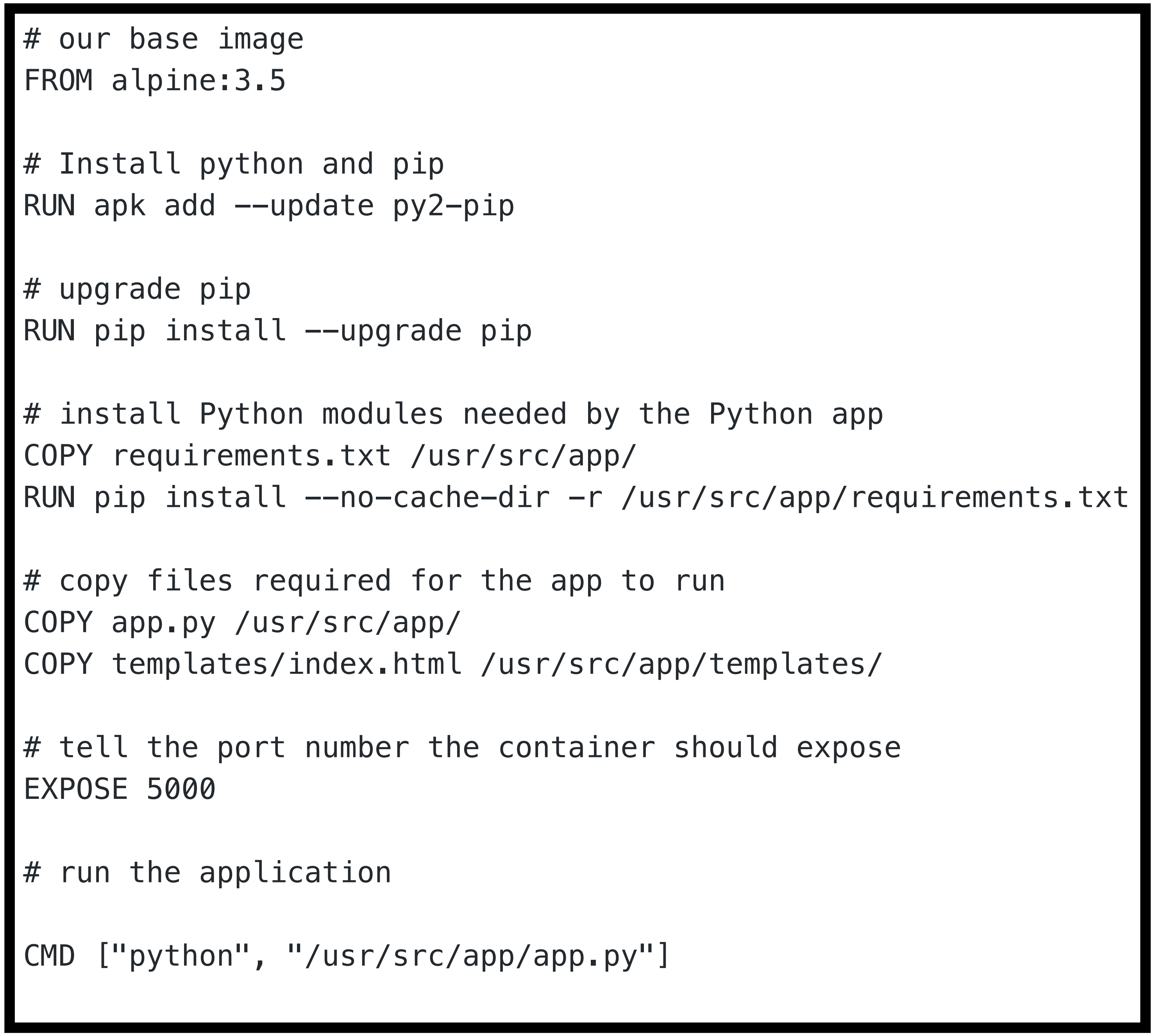
https://subscription.packtpub.com/book/application_development/9781788992329/1/ch01lvl1sec13/docker-container-architecture

Dockerfile

- A Dockerfile is a text file that contains all commands, in order, needed to build a given image

- It adheres to a specific format and set of instructions

- Using docker build, users can create an automated build that executes several command-line instructions in succession



```
# our base image
FROM alpine:3.5

# Install python and pip
RUN apk add --update py2-pip

# upgrade pip
RUN pip install --upgrade pip

# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

# copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

# tell the port number the container should expose
EXPOSE 5000

# run the application

CMD ["python", "/usr/src/app/app.py"]
```

https://docs.docker.com/engine/reference/builder/

https://docs.docker.com/samples/

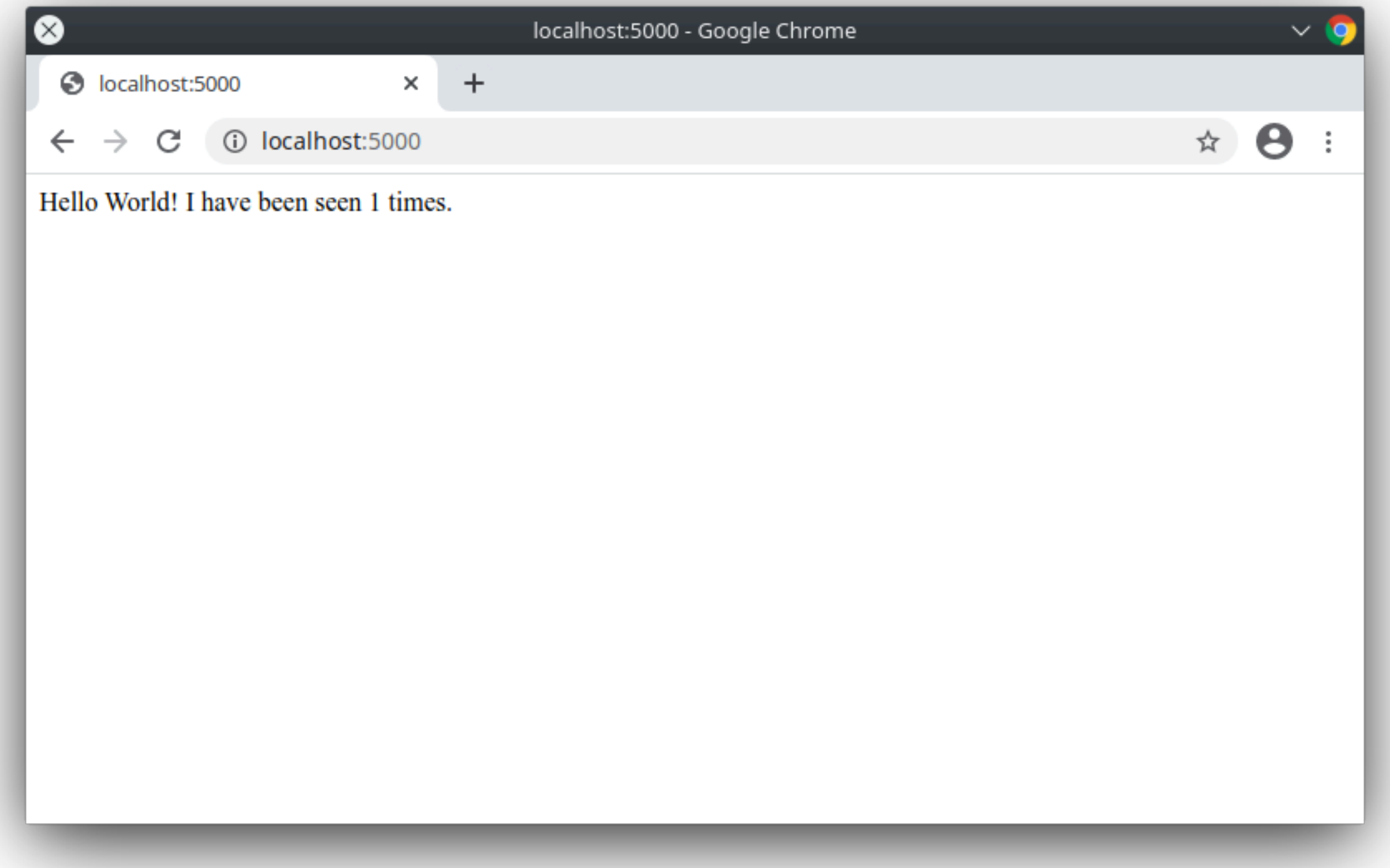
Docker-compose

- `docker-compose` can be viewed as an automated multi-container workflow

- It belongs to the Docker family, used to define and run multi-container applications

- With *compose*, we use a YAML file to configure application’s services and create all the app’s services from that configuration

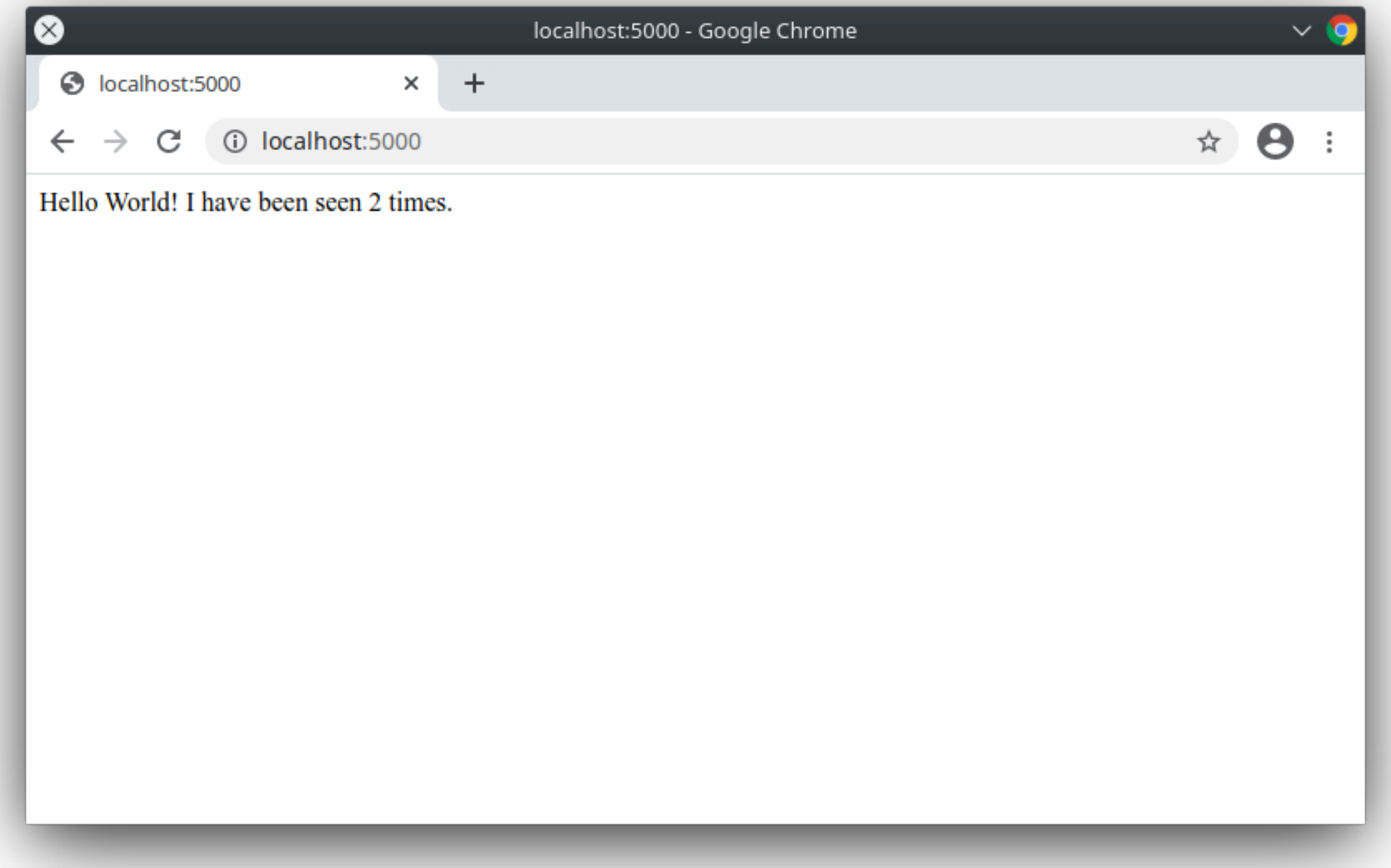
Docker-compose example



Hello World! I have been seen 1 times.

https://docs.docker.com/compose/gettingstarted/

Docker-compose example



Hello World! I have been seen 2 times.

https://docs.docker.com/compose/gettingstarted/

Docker-compose example

app.py

```
import time

import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)
```

https://docs.docker.com/compose/gettingstarted/

Docker-compose example

requirements.txt

```
flask
redis
```

Docker-compose example

Dockerfile

```
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

Docker-compose example

docker-compose.yaml

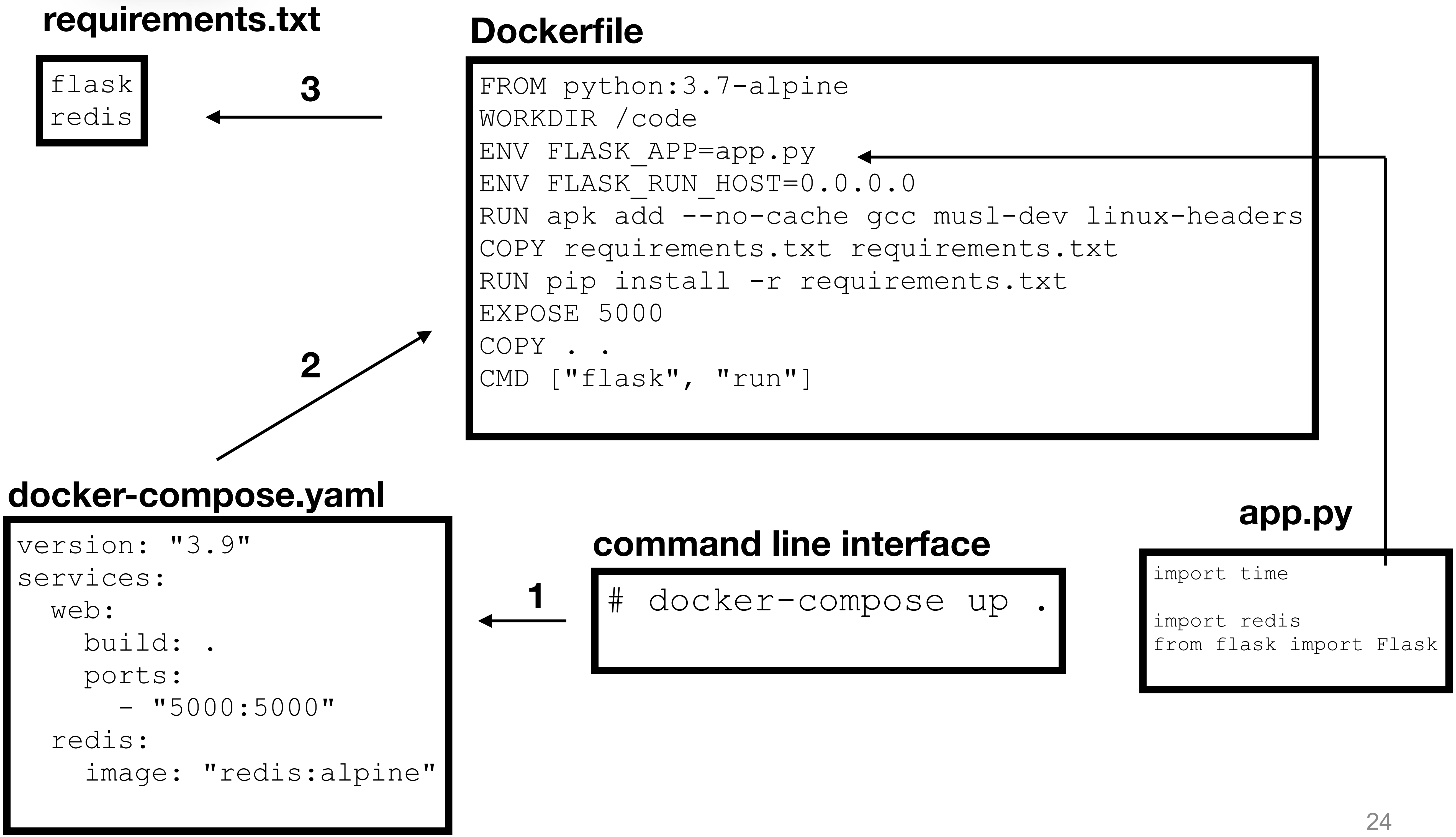
```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "5000:5000"
  redis:
    image: "redis:alpine"
```

Docker-compose example

command line interface

```
# docker-compose up .
```

Docker-compose example



requirements.txt

```
flask
redis
```

Dockerfile

```
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

docker-compose.yaml

```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "5000:5000"
  redis:
    image: "redis:alpine"
```

command line interface

```
# docker-compose up .
```

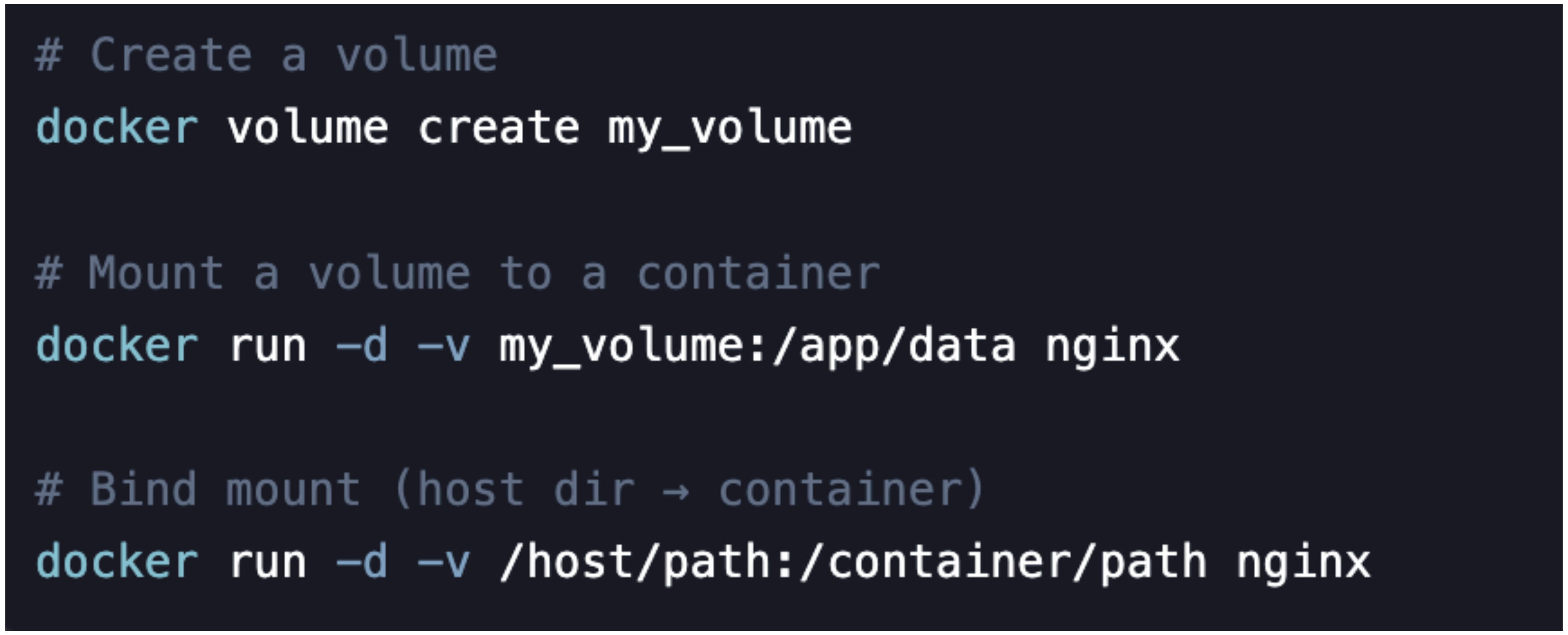
app.py

```
import time

import redis
from flask import Flask
```

Docker Storage

- Containers are ephemeral (data is lost when the container stops).

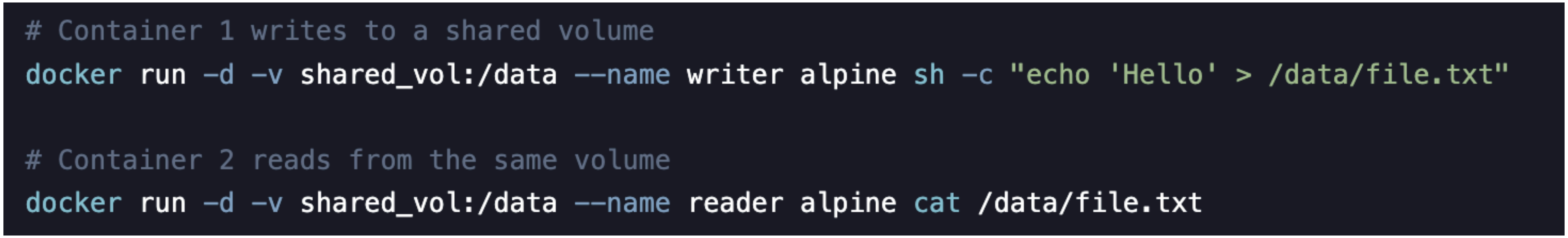
- Solutions for persistent data:
  - **Volumes:** Managed by Docker (stored in /var/lib/docker/volumes).
  - **Bind Mounts:** Link container paths to host machine directories.
  - **tmpfs Mounts:** Store data in host memory (non-persistent).



```
# Create a volume
docker volume create my_volume

# Mount a volume to a container
docker run -d -v my_volume:/app/data nginx

# Bind mount (host dir → container)
docker run -d -v /host/path:/container/path nginx
```

Docker Storage

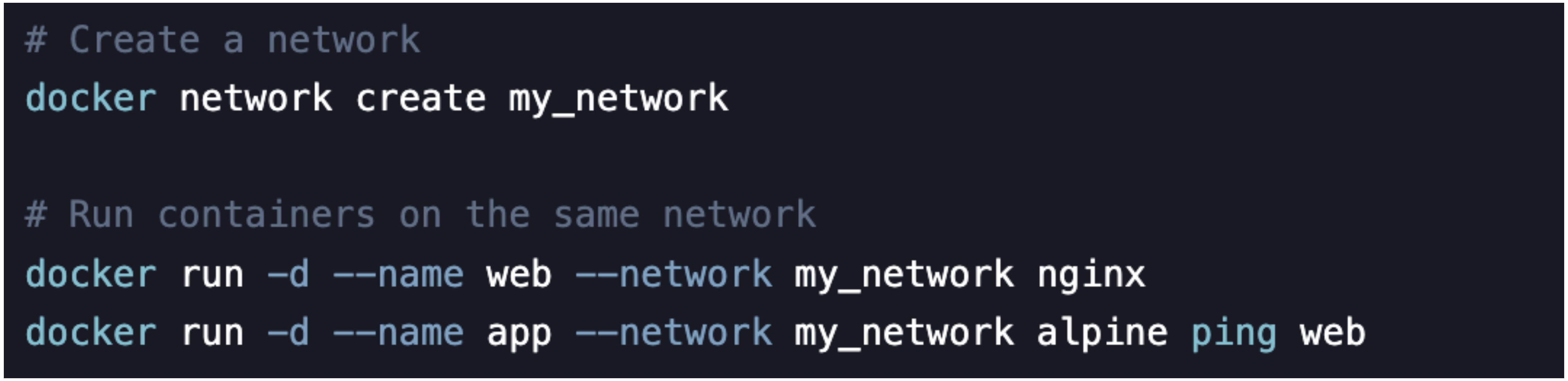
- Why Use Volumes?
  - Decouple storage from containers.
  - Back up/restore easily.
  - Share data between containers.



```
# Container 1 writes to a shared volume
docker run -d -v shared_vol:/data --name writer alpine sh -c "echo 'Hello' > /data/file.txt"

# Container 2 reads from the same volume
docker run -d -v shared_vol:/data --name reader alpine cat /data/file.txt
```

Docker Networking Basics

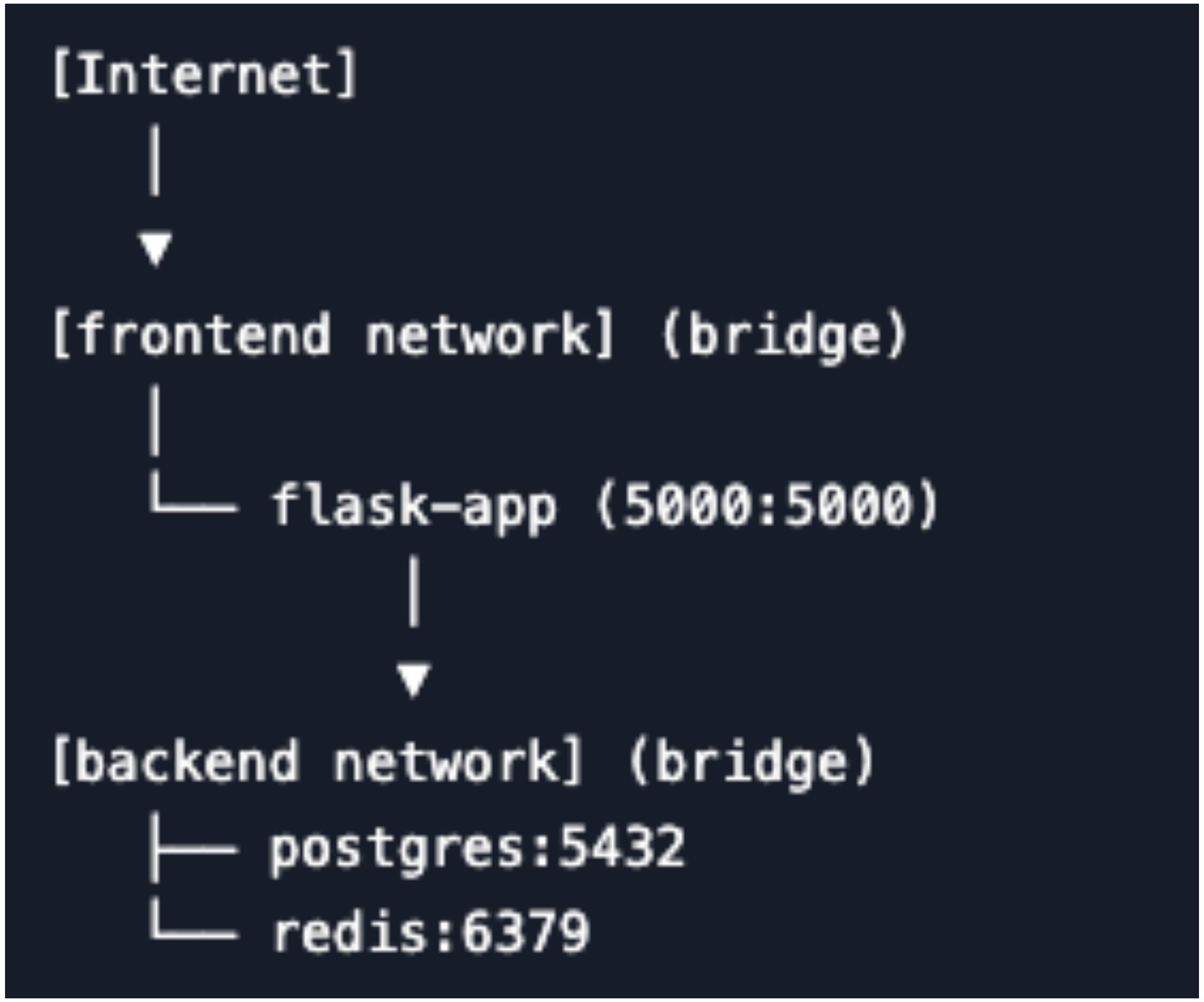
- Default Networks:
  - **Bridge:** Default network for containers (isolated communication).
  - **Host:** Bypasses Docker’s network isolation (container uses host’s network).
  - **None:** No networking (only loopback).



```
# Create a network
docker network create my_network

# Run containers on the same network
docker run -d --name web --network my_network nginx
docker run -d --name app --network my_network alpine ping web
```

Networking Use Case

- Multi-Service App with Isolated Networks

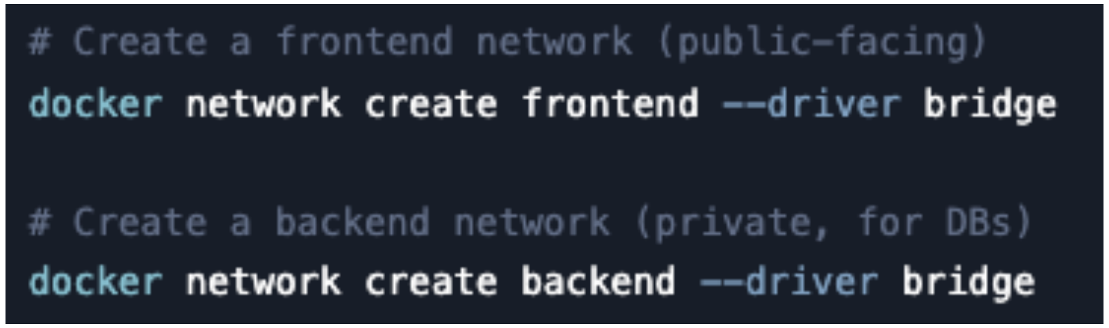


```
[Internet]
    |
    ▼
[frontend network] (bridge)
    |
    └── flask-app (5000:5000)
              |
              ▼
[backend network] (bridge)
    ├── postgres:5432
    └── redis:6379
```

Networking Use Case

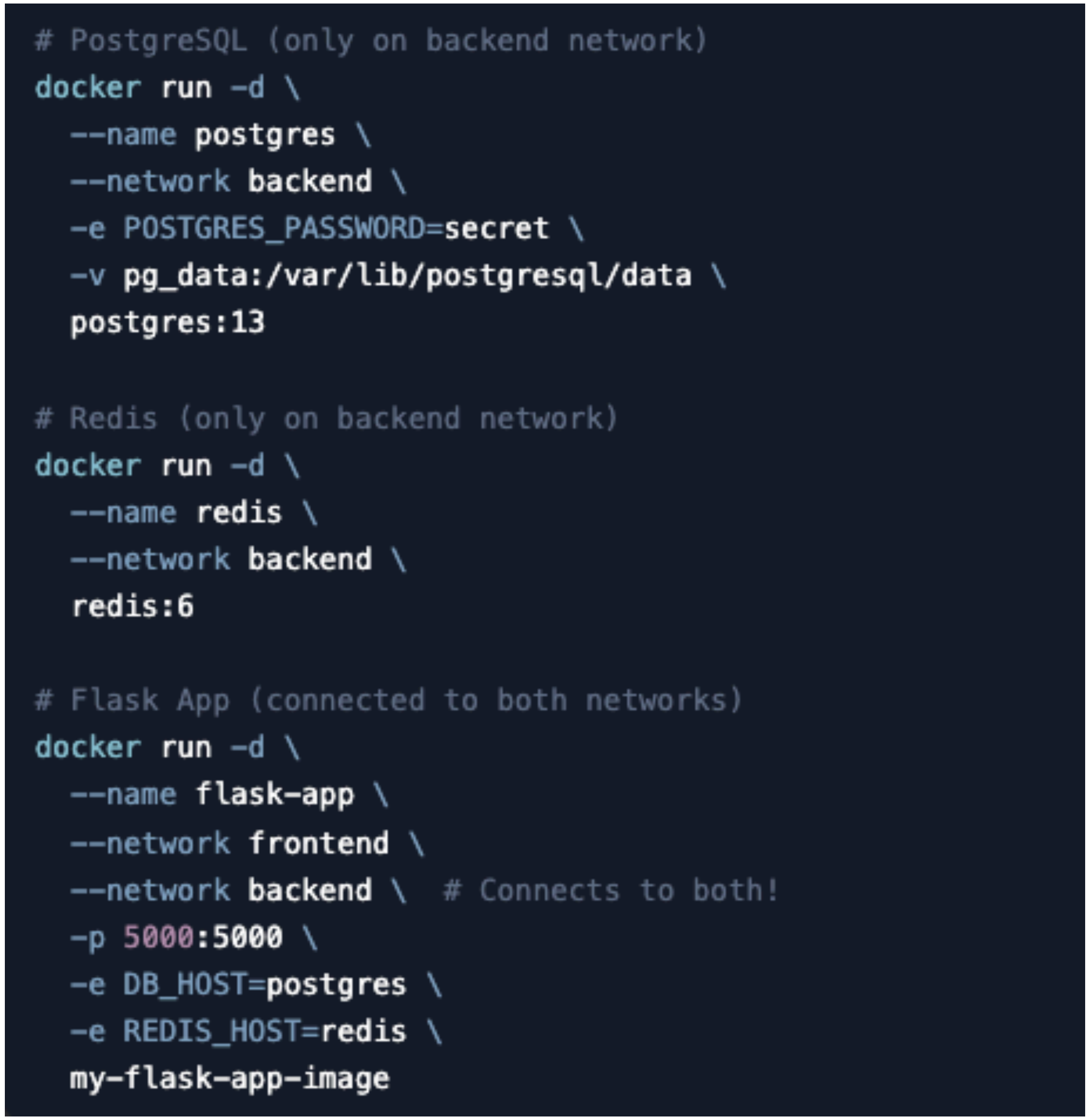
- Multi-Service App with Isolated Networks

  - Create a network.



```
# Create a frontend network (public-facing)
docker network create frontend --driver bridge

# Create a backend network (private, for DBs)
docker network create backend --driver bridge
```

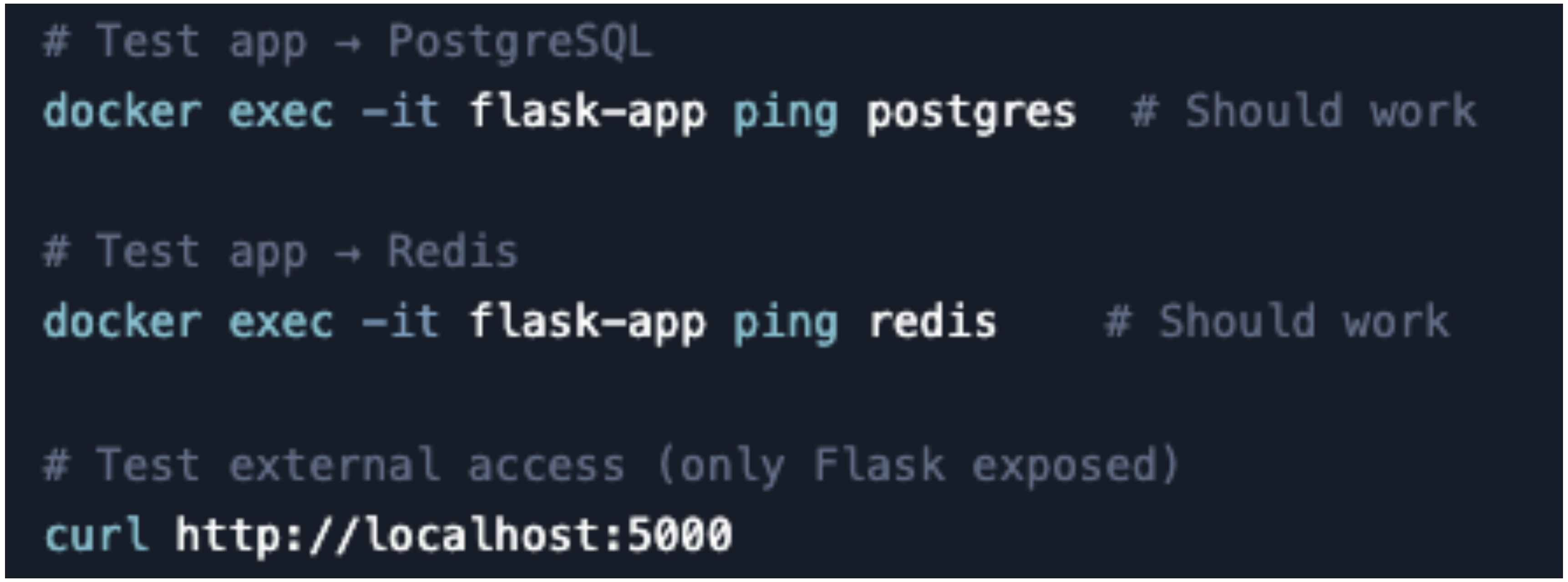
  - Services with Network Isolation



```
# PostgreSQL (only on backend network)
docker run -d \
  --name postgres \
  --network backend \
  -e POSTGRES_PASSWORD=secret \
  -v pg_data:/var/lib/postgresql/data \
  postgres:13

# Redis (only on backend network)
docker run -d \
  --name redis \
  --network backend \
  redis:6

# Flask App (connected to both networks)
docker run -d \
  --name flask-app \
  --network frontend \
  --network backend \  # Connects to both!
  -p 5000:5000 \
  -e DB_HOST=postgres \
  -e REDIS_HOST=redis \
  my-flask-app-image
```

Networking Use Case

- Multi-Service App with Isolated Networks
  - Verify Connectivity



```
# Test app → PostgreSQL
docker exec -it flask-app ping postgres  # Should work

# Test app → Redis
docker exec -it flask-app ping redis    # Should work

# Test external access (only Flask exposed)
curl http://localhost:5000
```

Docker Secrets

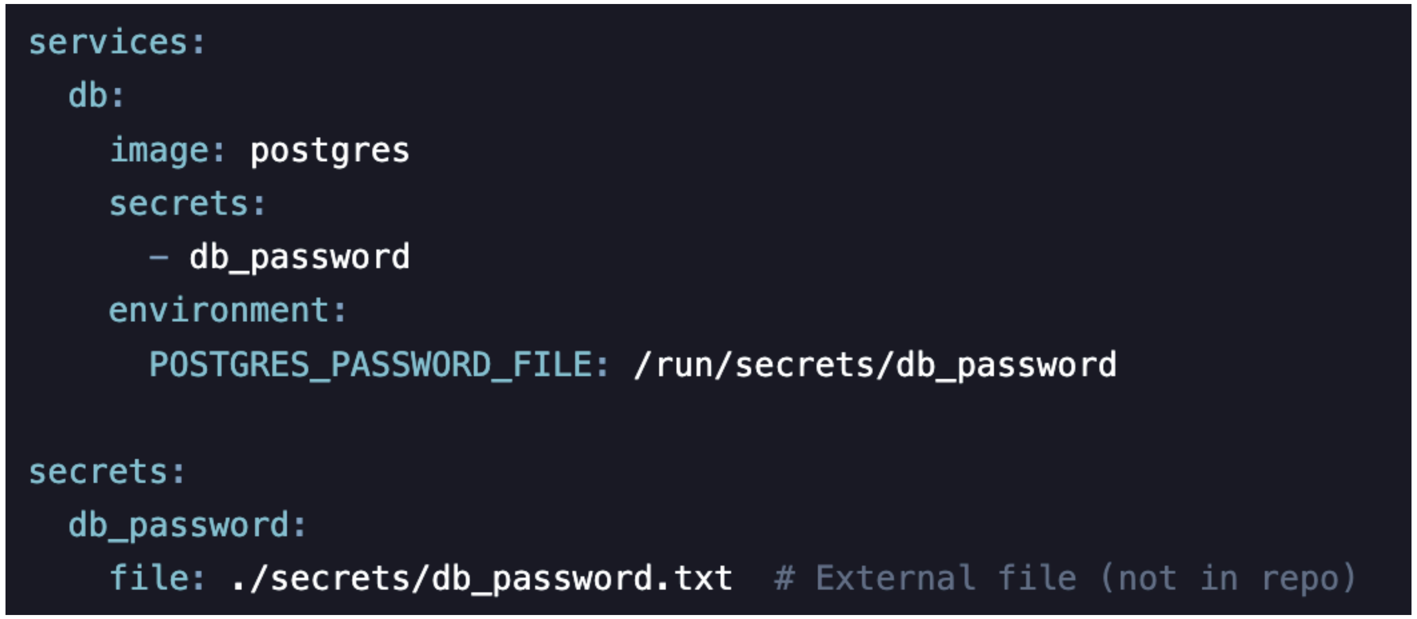
- Securing Sensitive Data in Docker

  - Problem:
    - Hardcoding secrets (passwords, API keys) in images or env files is unsafe.
    - Containers can leak secrets via logs or inspect tools.

  - Solutions:
    - Docker Secrets (Native for Swarm, but usable in Compose).
    - Environment Variables (Limited security).
    - Secret Mounts (Files injected at runtime).
    - Third-party Tools (HashiCorp Vault, AWS Secrets Manager).

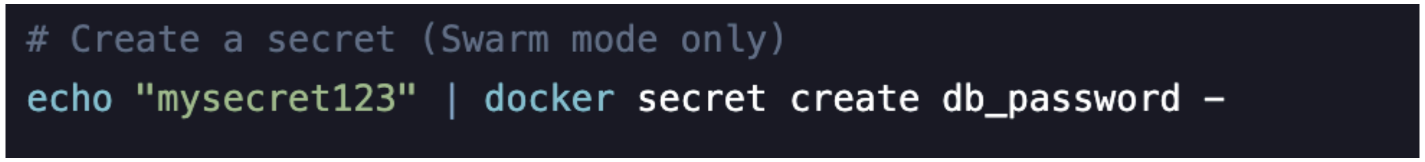
Docker Secrets

- Secrets in Docker
  - Docker Compose Example



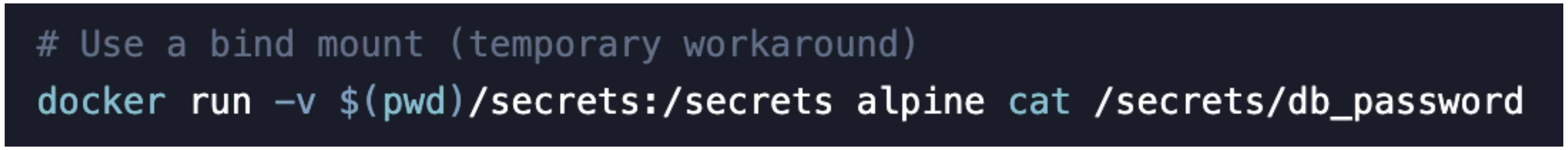
```yaml
services:
  db:
    image: postgres
    secrets:
      - db_password
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password

secrets:
  db_password:
    file: ./secrets/db_password.txt  # External file (not in repo)
```

  - Start PostgreSQL with Volume.



```
# Create a secret (Swarm mode only)
echo "mysecret123" | docker secret create db_password -
```

  - Connect an app container.



```
# Use a bind mount (temporary workaround)
docker run -v $(pwd)/secrets:/secrets alpine cat /secrets/db_password
```

Docker Secrets

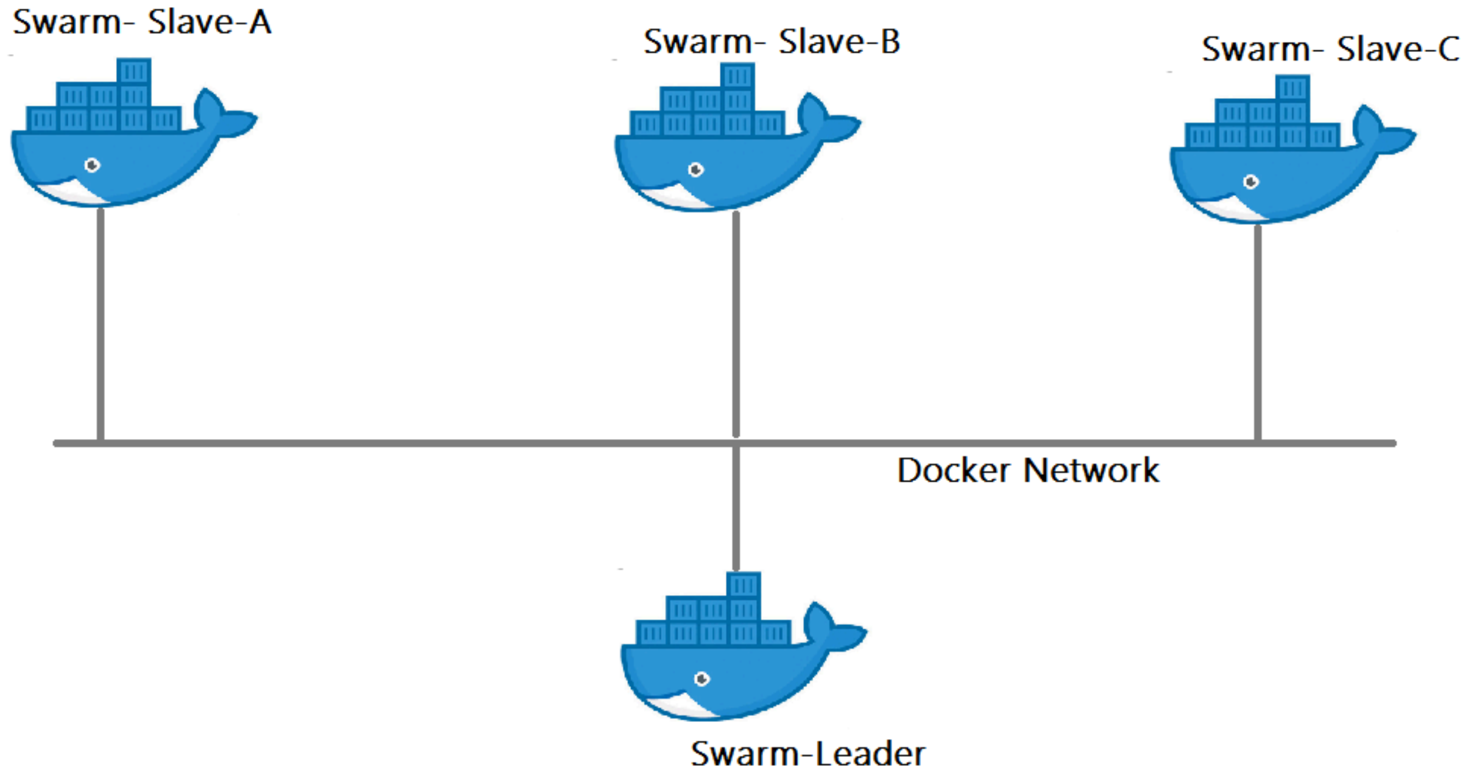
- Secrets in Docker
  - **Never** hardcode secrets in **Dockerfiles** or source code.
  - Prefer **Docker Secrets** or external tools (Vault) for production.
  - **Audit** secret usage (e.g., docker secret ls).

Container softwares

- OpenVZ
- Virtuozzo (Linux and Windows)
- Solaris Containers (Solaris)
- Spoon (Windows)
- VMware ThinApp (Windows)

Docker Swarm

Docker swarm is a container orchestration tool that allows users to manage multiple containers deployed across multiple host machines.



https://docs.docker.com/engine/swarm/
https://cloudnfv.wordpress.com/2019/09/10/docker-swarm/