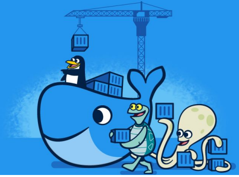


DevOps

Docker Containers

Containers

What is a Container ?

- Lightweight portable package with binaries allowing to encapsulate and run a program.

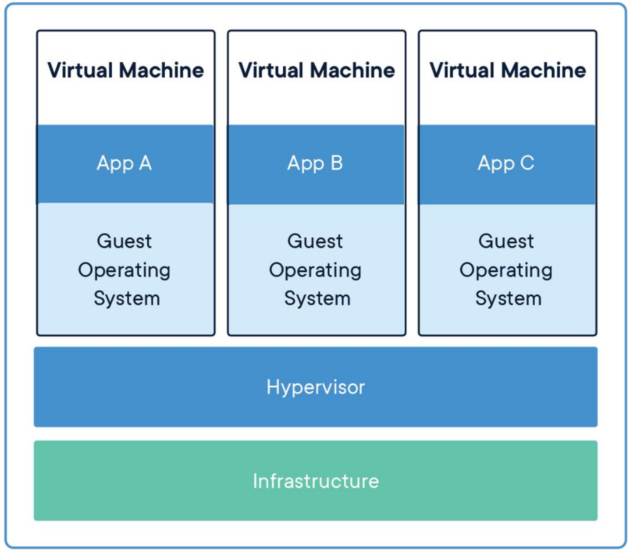
- Let’s say you want to run a program. All you need is :
    - A minimal base image allowing your to install / operate your program
    - A Runtime for the program
    - Dependencies
    - Your Application’s code

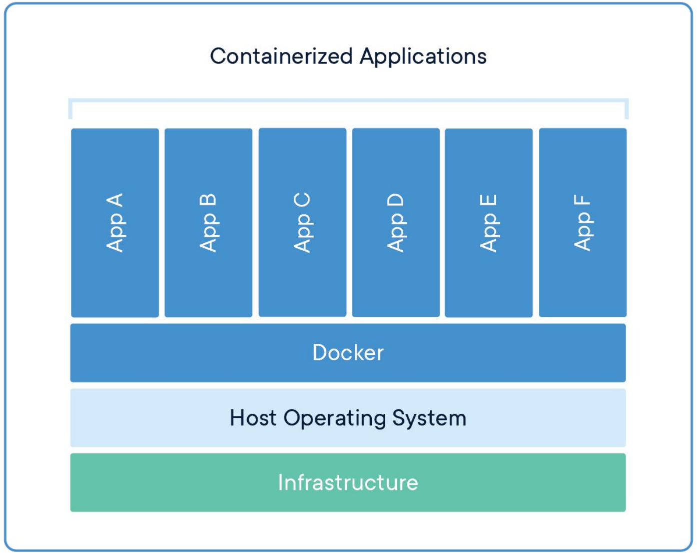
- You don’t need to bother with your Kernel and what’s installed on it.

- In a project, containers are very practical deliverable since they can be run “anywhere”

Containers

Containers vs VMs ?





source : docker.com

- Hardware level
- Whole OS
- Reserves resources
- Slow to start

- Lightweight
- Share computer’s OS & resources
- Start very fast

Containers

Why Containers / Advantages

- Start instantly
- Easy to deploy & run
- Easy to reproduce problems
- Easy to scale & distribute
- Built from a declarative state & reproductible
- Many available tools ready to use (public images)
- Platform independent

Containers

Orchestration

- Once we have containers, it is essential to:
    - Handle auto-scaling / replication.
    - Check their health & restart them.
    - Ensure they can communicate together and access persistent storage volumes.

- Container Orchestration tools come to the rescue:
    - Mesosphere Marathon
    - Kubernetes
    - Docker Swarm

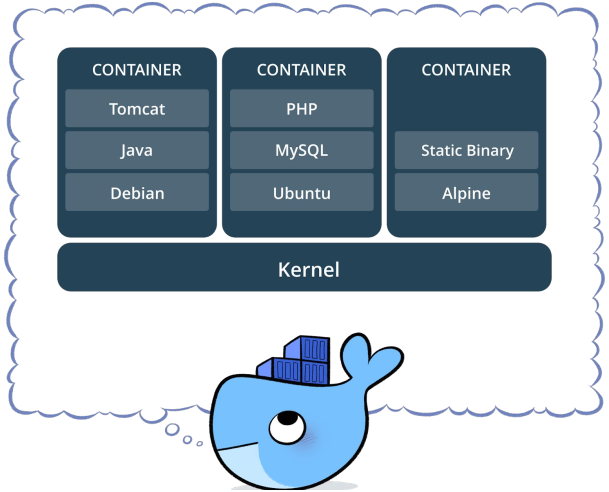
Reminders on Containerization

- Containers allow deploying multiple services on hosts while isolating them from each other and from hosts packages.

- Consistent way to deliver and run applications

- Fast deployment, easy to scale

- Offer modularity to operate and maintain services



Reminders on Containerization

Best practices with Containers:

- Each container should do one thing and do it well.
- Containers should be as light as possible.
- Users should not have too much privileges (disable root user).
- Scan non-official images before running them.

Docker

- Created for dotCloud’s needs by Solomon Hykes
- Distributed as an Open Source project since 2013
- Written in Go
- Available on Linux, macOS and Windows since 2020 (Windows Subsystem for Linux 2)

Docker Components

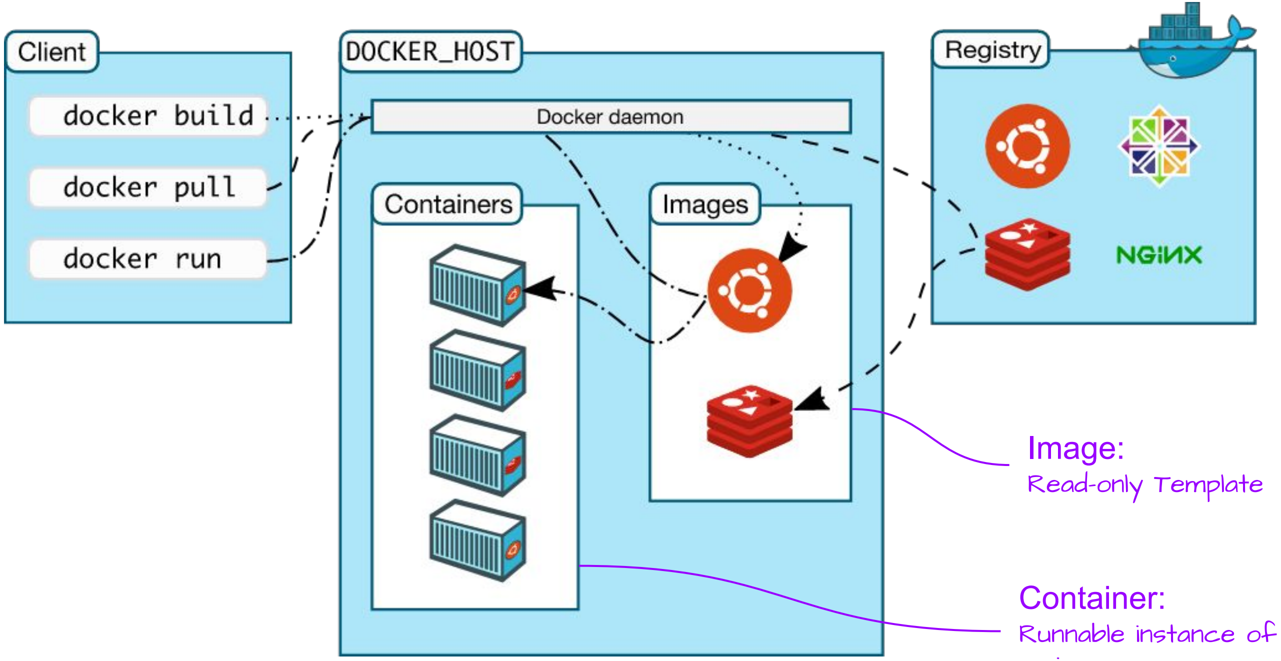
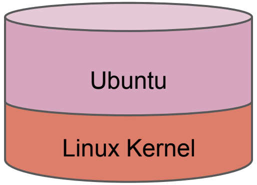


Image:
Read-only Template

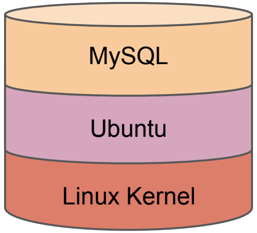
Container:
Runnable instance of
an image

Docker Images

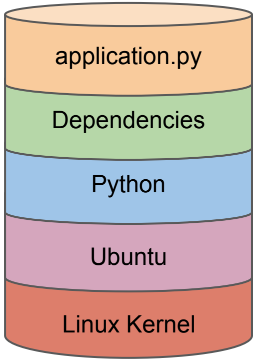
- Images are built with stacked layers (increments)



Ubuntu Image



MySQL on
Ubuntu Image



Some Python app
on Ubuntu Image

Docker Images

- Images are fully-named and referenced with :
  - an optional prefix, the registry hostname
  - a name identifying the image
  - a tag (version), defaulting to “latest”

nginx refers to the latest tag for the official nginx image.

elasticsearch:7.14.1 refers to the official elasticsearch image with version 7.14.1

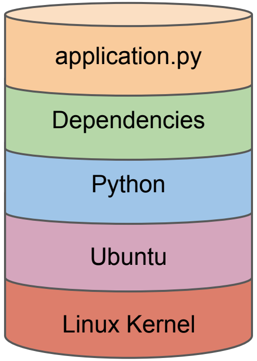
bitnami/kafka:2.8.1 refers to Kafka version 2.8.1 built by bitnami

Docker Images

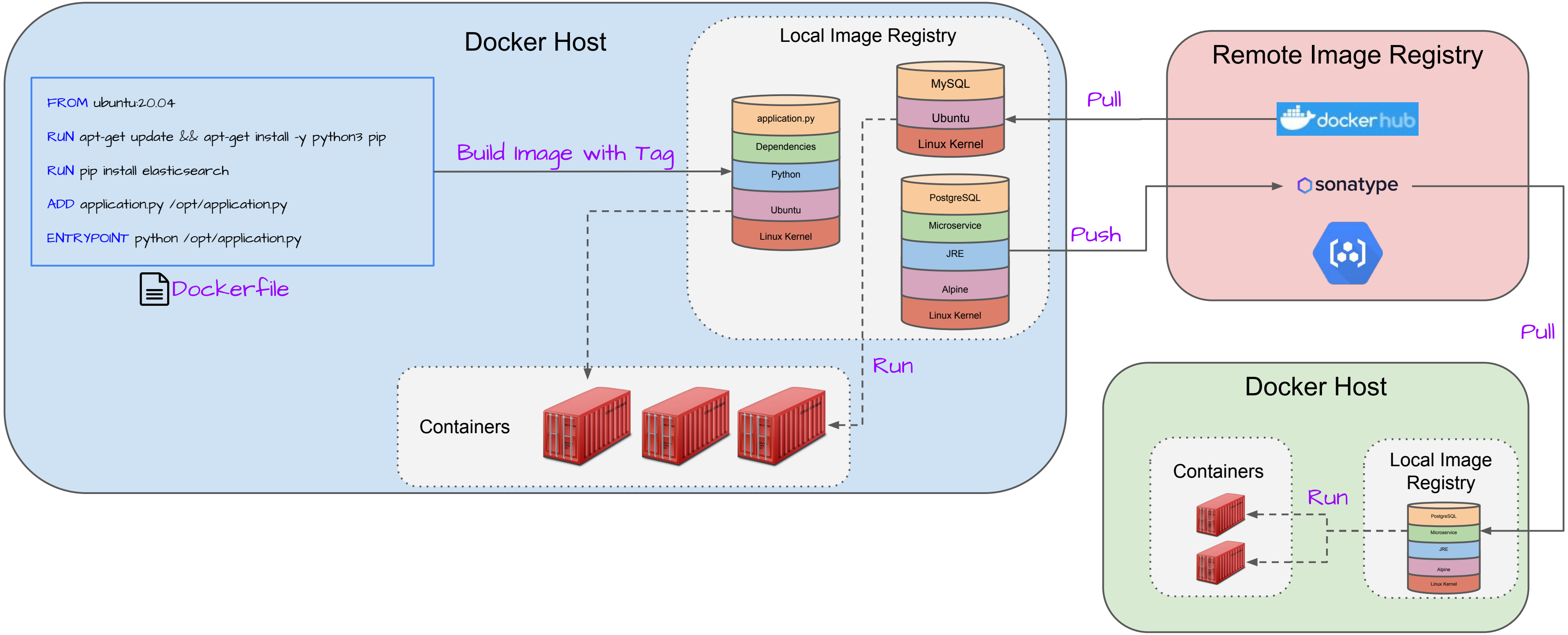
- Images are built using a base image and performing actions on it.
- Either described in Dockerfiles or built by committing changes to an image.

Image name with tag

```
FROM ubuntu:20.04

RUN apt-get update && apt-get install -y python3 pip

RUN pip install elasticsearch

ADD application.py /opt/application.py

ENTRYPOINT python /opt/application.py
```

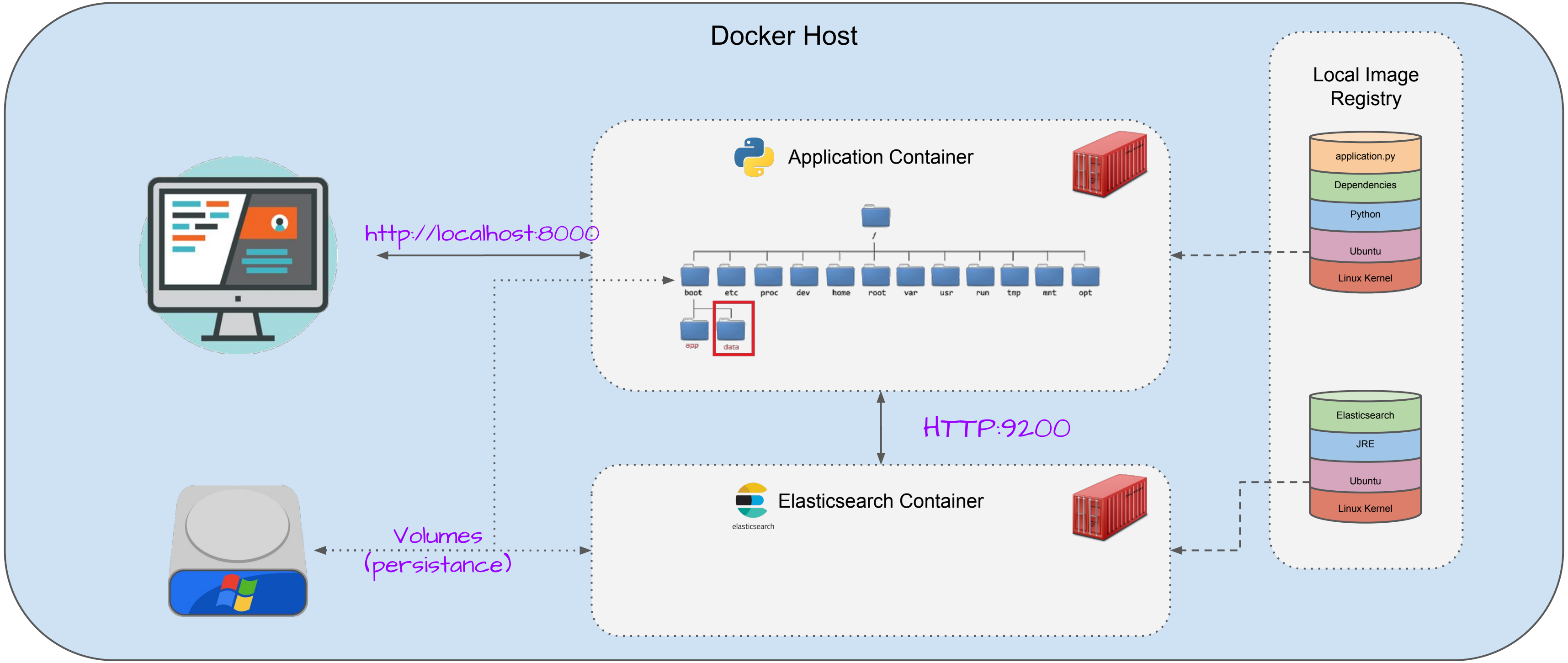


Docker Images



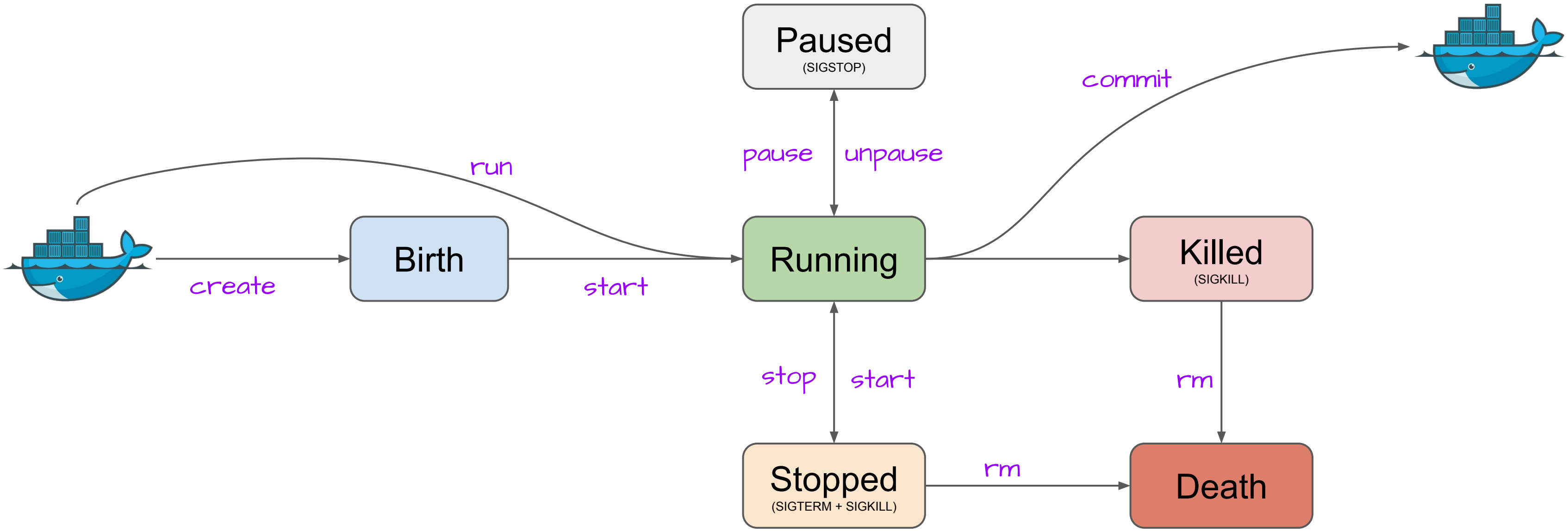
```
FROM ubuntu:20.04
RUN apt-get update && apt-get install -y python3 pip
RUN pip install elasticsearch
ADD application.py /opt/application.py
ENTRYPOINT python /opt/application.py
```

Docker Containers



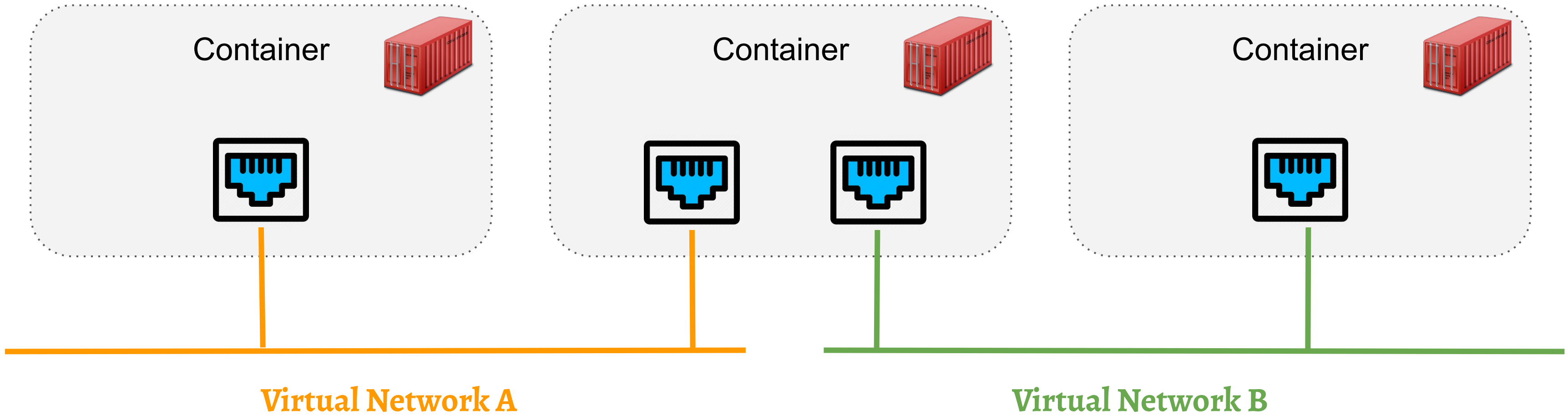
Docker Containers

Lifecycle



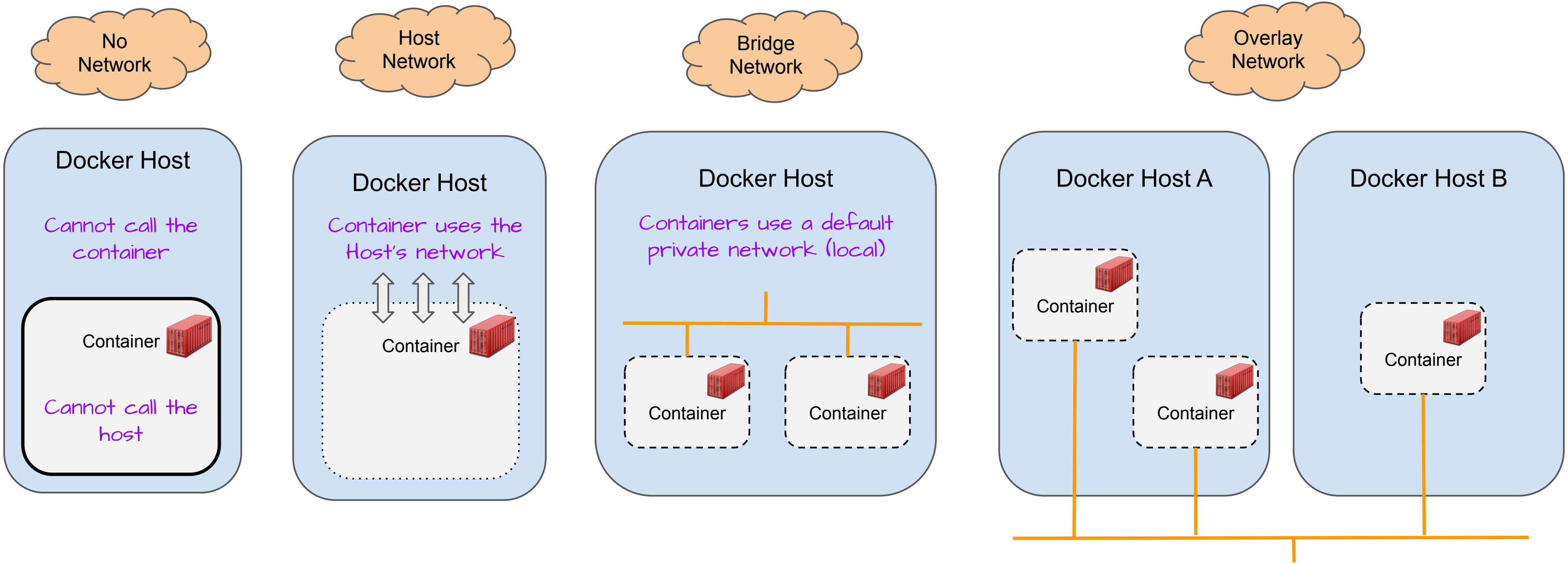
Docker Network

- Docker comes with the ability to create Virtual Networks:
  - Allow you to interact with Containers
  - Allow containers to communicate



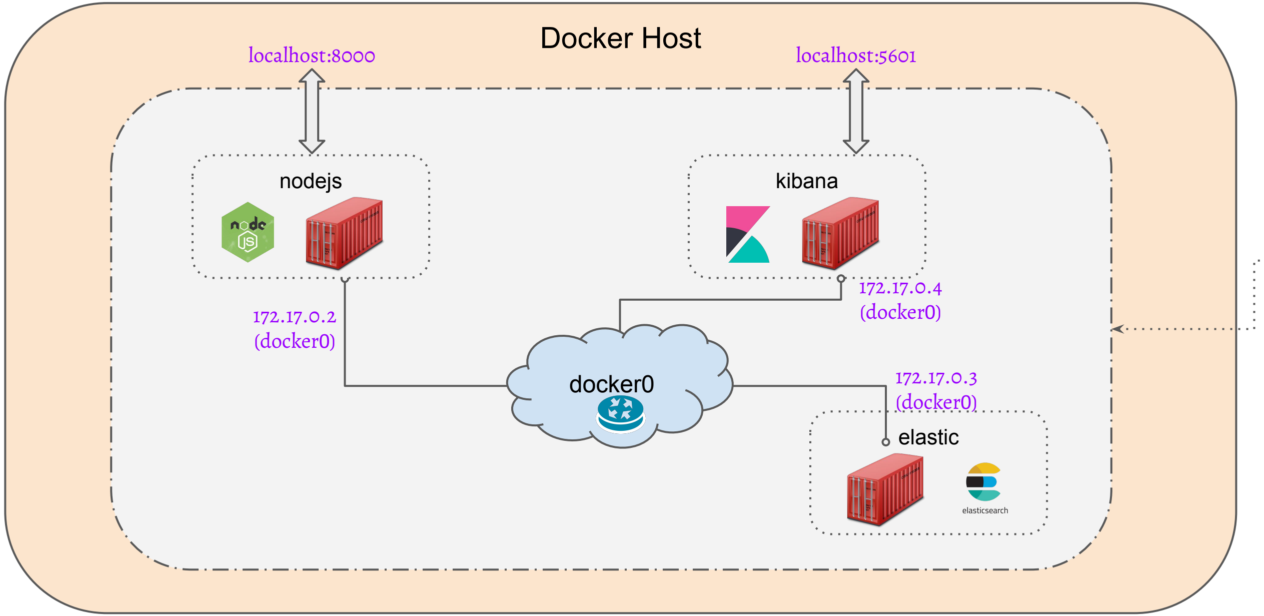
Docker Network

- Multiple network isolation possibilities : nothing / everything / virtual networks
- Allow exposing / mapping specific ports



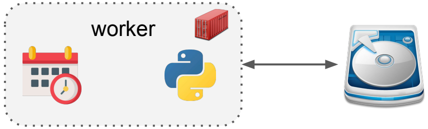
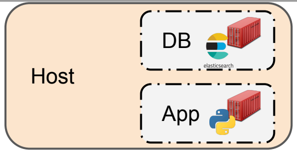
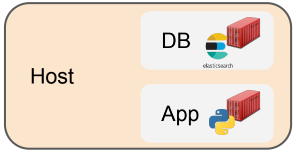
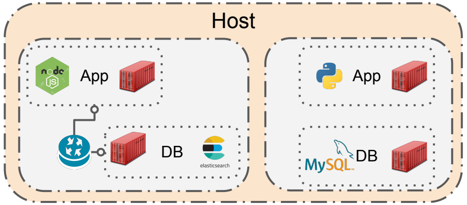
Docker Network

- Containers can call each other using their names using an internal DNS.



```
{
   "Name": "my-network",
   "Driver": "bridge",
   "IPAM": {
      "Driver": "default",
      "Config": [
         {
            "Subnet": "172.17.0.0/16",
            "Gateway": "172.17.0.1"
         }
      ]
   },
   "Containers": {

"35a42839b835391bb4ba021ffd1d151593a176
63edd75f7d64563fb65c025037": {
         "Name": "nodejs",
         "MacAddress": "02:42:ac:12:00:02",
         "IPv4Address": "172.17.0.2/16"
      },

"c0a0f3cc90da4372ad2f3f82cf2a7cfda4f64c7f2
513170f870792823b068d97": {
         "Name": "elastic",
         "MacAddress": "02:42:ac:12:00:03",
         "IPv4Address": "172.17.0.3/16"
      }
   }
}
```

Docker Network

Network Mode	Description	Example Use case
none	No networking. Cannot be called from outside and cannot call external services.	Standalone worker 
bridge	**Default**. Creates virtual interfaces to allow communication between the containers within the network. This virtual network is shared with Host.	We may need to : - Connect containers together - Expose specific ports - Access the external world 
host	Containers use the Host's network. Everything's available from inside and outside. There is **no network-level isolation**.	All we're interested in is the environment isolation (binaries / packages & versions). Services are managed at the container level. 
overlay	Create a Virtual network allowing containers to see each other across the L2 physical network.	Containers: - Are in a same subnet. - Can access external world - Can be distributed across multiple nodes (orchestration)  Host node JS App App DB elasticsearch MySQL DB

Docker

Testing with Containers

- Images exist for multiple services.
- Possible to create custom images.
- Containers are lightweight and can be run everywhere.
- Containers allow to **reproduce** things.
- Containers can be used for Integration tests.

https://www.testcontainers.org/

Docker Commands

Images

Command	Description	Examples
docker pull <image name>[:tag]	Downloads an image with a specific tag (latest by default).	docker pull ubuntu docker pull elasticsearch:7.14.1
docker build	Builds an image using a Dockerfile.	docker build -t myapp:0.1 .
docker images	Lists images with tags available in the local image registry.	docker images docker images --all
docker commit <containerId> <image>	Creates a new image from a container.	docker commit 597ce1600cf4 custom-ubuntu:20.04
docker rmi <image>[:tag]	Removes an image from the local registry.	docker rmi myapp:0.1
docker push <image name>[:tag]	Publishes an image from the local registry to a remote.	docker image push registry-host:port/myapp:0.1
docker save	Exports the image to a tar archive.	docker save myapp:0.1 > myapp_0-1.tar
docker load	Imports an image from a tar archive.	docker load < myapp_0-1.tar

Docker Commands

Containers

Command	Description	Examples
docker run <image name>[:tag]	Creates a container from an image.	docker run -it ubuntu:20.04 docker run -e "discovery.type=single-node" -d -p 9200:9200 elasticsearch:7.14.1 docker run -v /opt/app:/app --link elastic:elastic faucet/python3 pyhton /app/main.py
docker exec	Uses a container and runs actions on it.	docker exec -it 9e8eb83f233f ls /app
docker ps	Lists containers on host	docker ps -a
docker logs <containerId>	Shows the console logs for a specific container.	docker logs 9e8eb83f233f
docker stop <containerId>	Stops the specified container(s).	docker stop 9e8eb83f233f ad14edb7724a
docker rm <containerId>	Removes the specified container(s). The container(s) must be stopped or killed before they can be removed.	docker rm 9e8eb83f233f ad14edb7724a

Dockerfile Instructions

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

Instruction	Description	Examples
FROM <image>[:tag]	Used to specify a base image. Any Dockerfile must start with a FROM instruction, unless arguments are required.	**FROM** ubuntu:20.04
RUN <cmd> [args]	Executes any command / program available at this point. May also include arguments.	**FROM** ubuntu:20.04 **RUN** apt-get update && apt-get install python3 **RUN** python3 myscript.py
ARG	Allows to define arguments, and optionally affect a default value. Build-time variables. If no value set, the image won't be built.	**ARG** UBUNTU_VERSION=20.04 **FROM** ubuntu:$UBUNTU_VERSION
ENV	Defines environment variables. Available both at build and run time.	**FROM** ubuntu:20.04 **ADD** application /opt/application **ENV** HTTP_PORT=8000
ADD	Lets you copy files from your host (or an URL) to a specified destination path.	**FROM** ubuntu:20.04 **ADD** application /opt/application **ADD** https://someserver.com/archive.tar.gz /opt/
EXPOSE <port>	Informs Docker that the containers should listen at the specified ports. It does not actually "publish" the port at runtime.	**FROM** ubuntu:20.04 **ADD** application /opt/application **EXPOSE** 8000
ENTRYPOINT	Specifies what will be executed when creating a container.	**FROM** ubuntu:20.04 **ADD** application /opt/application **ENTRYPOINT** /opt/application/main.py 8000

Dockerfile

Advices

- Use official images whenever possible:
  - Clear documentation
  - Maintained
  - Secure
  - Community

- Make images as small as possible:
  - Start with minimal base images (ie: alpine linux)
  - Reduce the number of layers when possible (ie: combine RUN instructions)

- Make images build fast:
  - Each instruction creates an intermediate layer in your cache
  - Order your Dockerfile instructions and start with the layers (instructions) which rarely change.

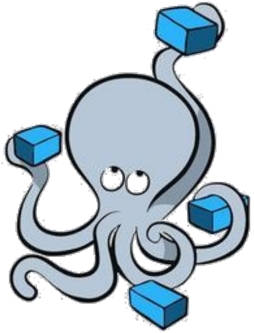
- Images should be pre-configured but customizable (as much as possible)
  - Easy to start
  - Easy to override specific configurations

- Use the .dockerignore file to exclude irrelevant files

Docker Compose

1. Command Line Interface tool

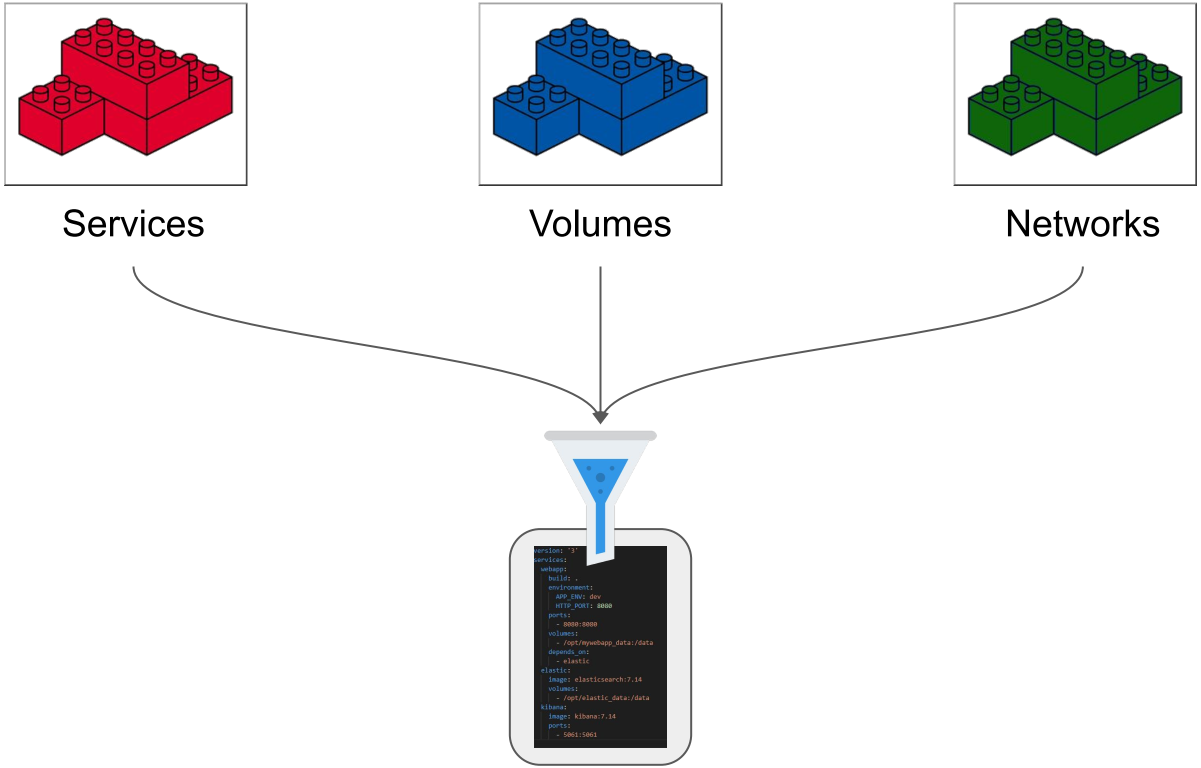
2. Define and run multi Docker-container systems



3. No more tedious docker run commands

4. Easy to run reproductible / complex systems

Docker Compose



Services

Volumes

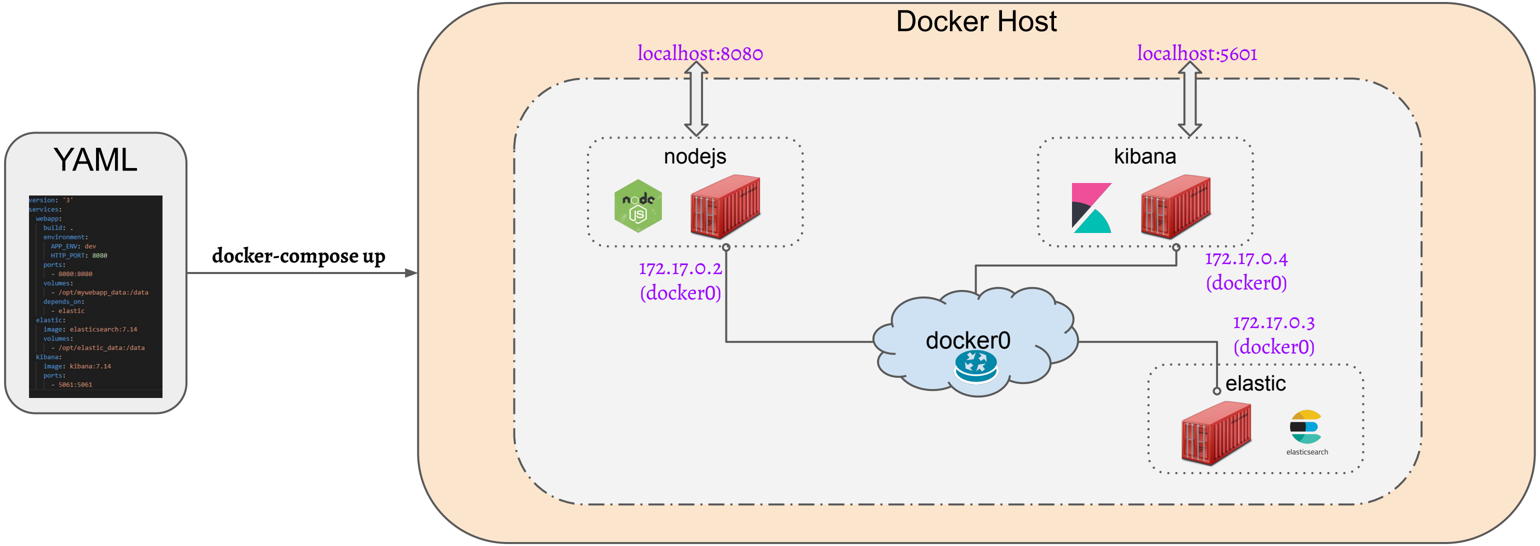
Networks

```yaml
version: '3'
services:
  webapp:
    build: .
    environment:
      APP_ENV: dev
      HTTP_PORT: 8080
    ports:
      - 8080:8080
    volumes:
      - /opt/mywebapp_data:/data
    depends_on:
      - elastic
  elastic:
    image: elasticsearch:7.14
    volumes:
      - /opt/elastic_data:/data
  kibana:
    image: kibana:7.14
    ports:
      - 5061:5061
```

Docker Compose

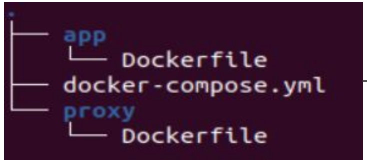
- IaC : Describe resources & services and how they integrate with each other in a single Yaml file (containers, volumes, networks).

- Very useful for development environments to start / share / operate multi-container applications.

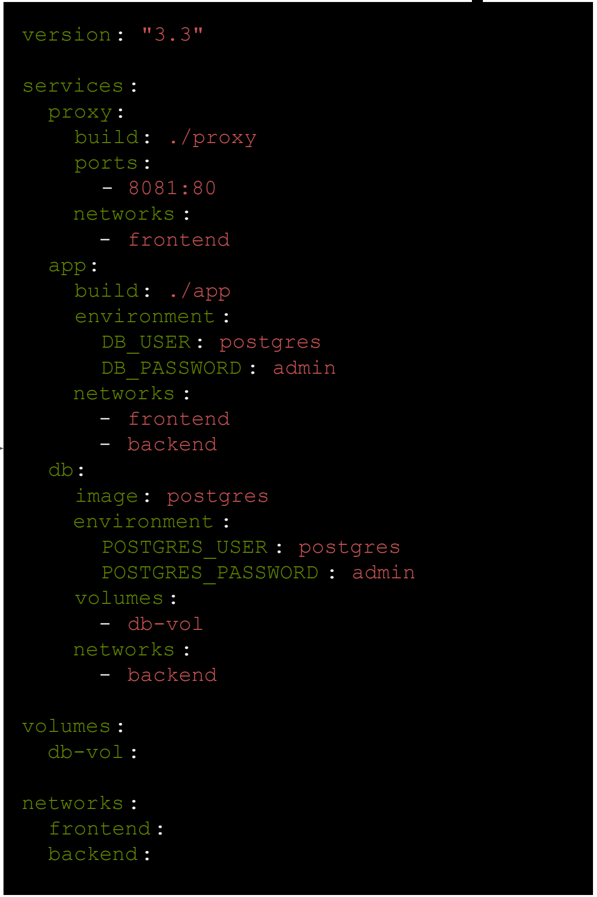


```yaml
version: '3'
services:
  webapp:
    build: .
    environment:
      APP_ENV: dev
      HTTP_PORT: 8080
    ports:
      - 8080:8080
    volumes:
      - /opt/mywebapp_data:/data
    depends_on:
      - elastic
  elastic:
    image: elasticsearch:7.14
    volumes:
      - /opt/elastic_data:/data
  kibana:
    image: kibana:7.14
    ports:
      - 5061:5061
```

Docker Compose



```
.
├── app
│   └── Dockerfile
├── docker-compose.yml
└── proxy
    └── Dockerfile
```



```
version: "3.3"

services:
  proxy:
    build: ./proxy
    ports:
      - 8081:80
    networks :
      - frontend
  app:
    build: ./app
    environment :
      DB_USER: postgres
      DB_PASSWORD : admin
    networks :
      - frontend
      - backend
  db:
    image: postgres
    environment :
      POSTGRES_USER : postgres
      POSTGRES_PASSWORD : admin
    volumes:
      - db-vol
    networks :
      - backend

volumes:
  db-vol:

networks:
  frontend:
  backend:
```

Docker Compose

- Interesting to use variables in compose files:
  - Instead of repeating values across the files (ie: DB user/password)
  - To externalize values which may change (ie: image versions)

- Compose files can include variables. This allows us to inject configuration when starting services.

- This can be achieved with .env files

```
version: '3'
services:
  db:
    image: mysql:latest
    environment:
      - MYSQL_DATABASE=${MYSQL_DB}
      - MYSQL_USER=${MYSQL_USER}
      - MYSQL_PASSWORD=${MYSQL_PW}
```

docker-compose.yml

```
MYSQL_DB=db
MYSQL_USER=my_username
MYSQL_PW=K6c574T@9Qd
```

.env

```
version: '3'
services:
  db:
    image: mysql:latest
    env_file: .env
```

docker-compose.yml

```
MYSQL_DATABASE=db
MYSQL_USER=my_username
MYSQL_PASSWORD=K6c574T@9Qd
```

.env

Docker Swarm

- Docker Compose : Tool used to manage multi-container applications on a **single machine** only.

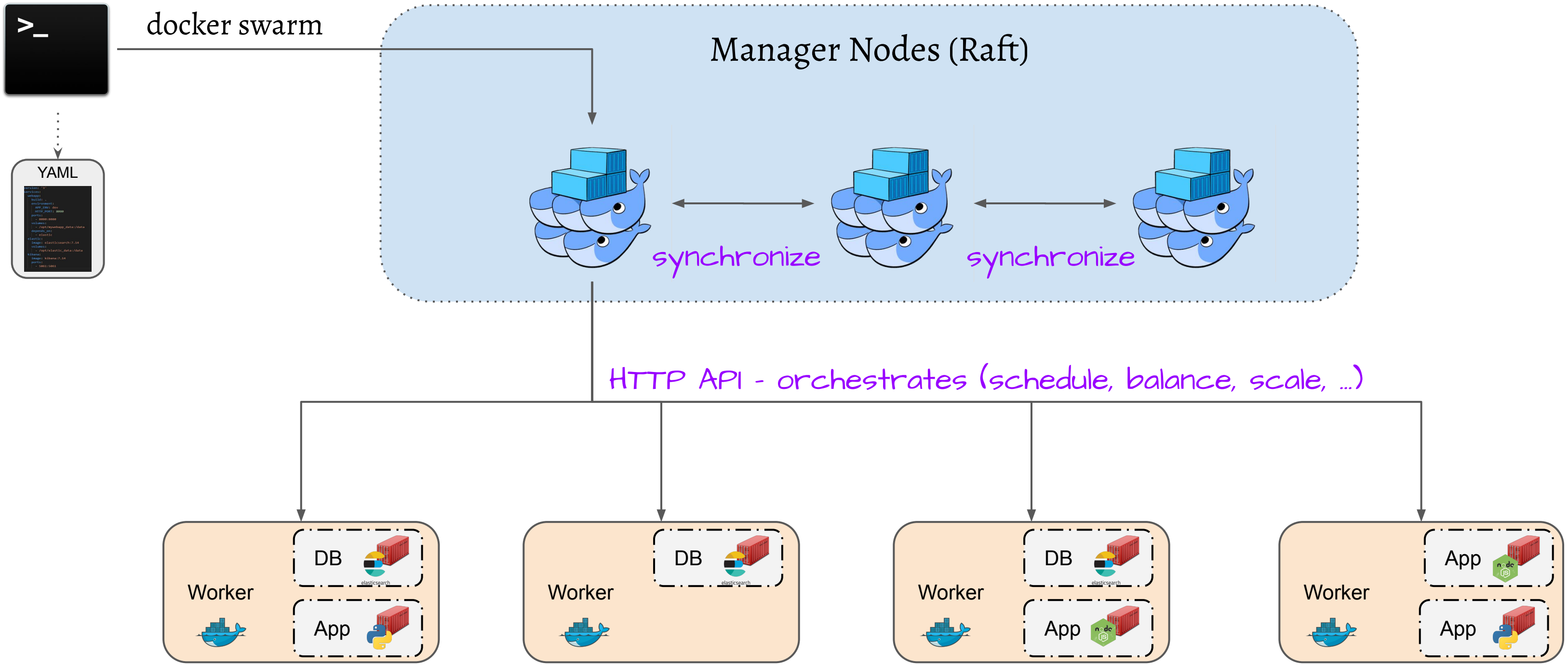
- Docker Swarm : Tool designed to manage multi-container applications on **clusters**. Acts as a container <u>**orchestrator**</u>.

- Both Compose & Swarm can use a same docker-compose file.

- Some features may be limited to either Compose or Swarm.

    For instance, secrets may be defined in Swarm but would be ignored in Compose.

Docker Swarm


docker swarm
YAML
Manager Nodes (Raft)
synchronize
synchronize
HTTP API - orchestrates (schedule, balance, scale, ...)
Worker
DB
App
Worker
DB
Worker
DB
App
Worker
App
App