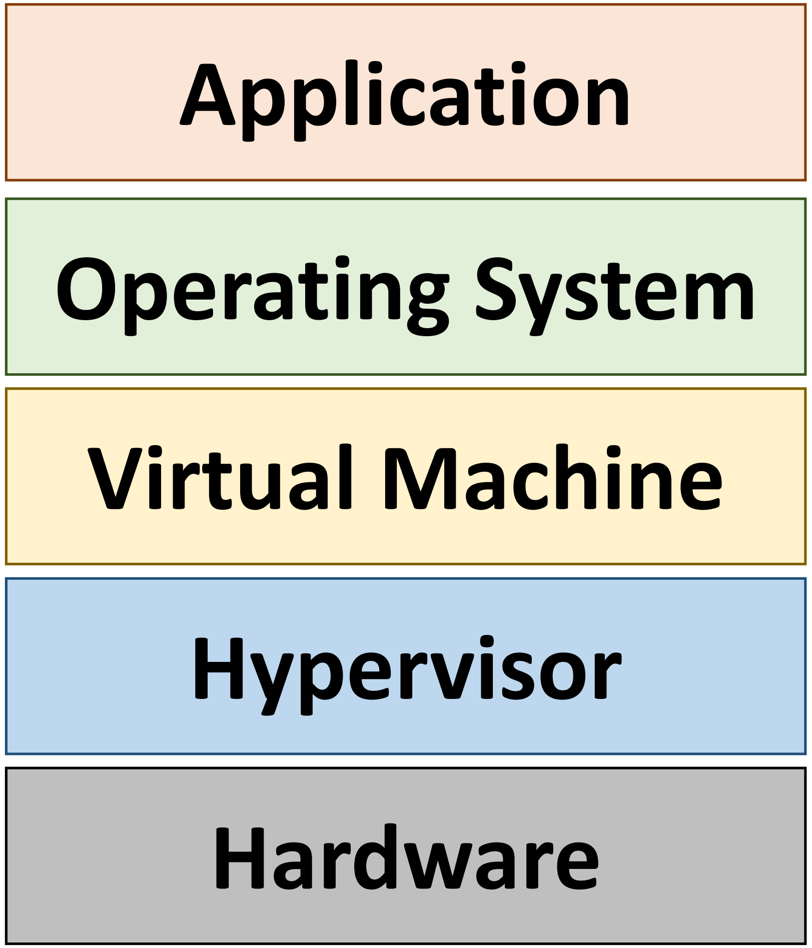
Containers

Docker

Hypervisor and Virtual Machines

- Hypervisor is a process that separates the operating system from the hardware
  - Also know as Virtual Machine monitor (VMM)
- Virtual machines is a software that emulates a computer system



What is Virtualization?

- Technology that allows you to run multiple operating systems on a single computer concurrently
  - Using hypervisor and virtual machines
- Typical computer setup
  - Single OS per machine
  - Run multiple applications on the same machine – may have dependencies conflict
  - Under utilize resources

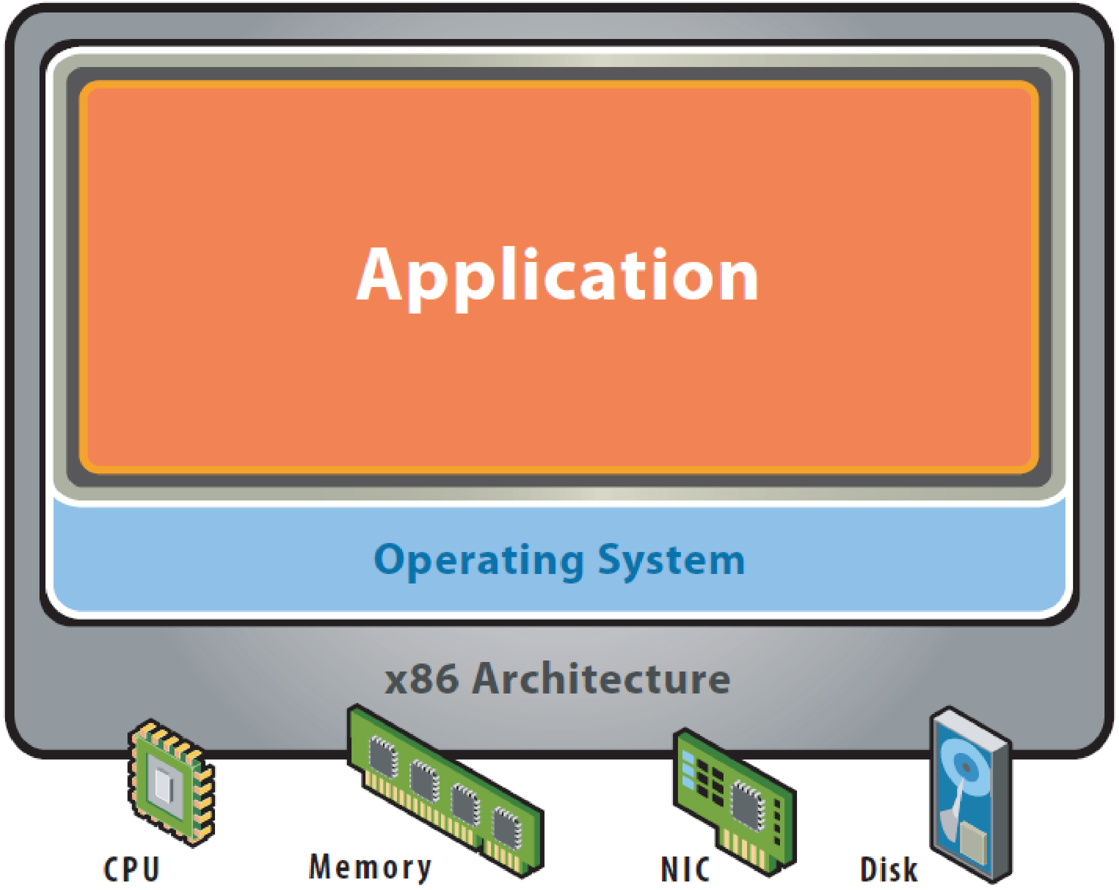


Image from https://www.vmware.com/pdf/virtualization.pdf

Benefits of Virtualization

- Hardware independent from the operating system
- Manage OS and application as a single unit by encapsulating them in a virtual machine
- Increase the system's utilization
  - By allowing 2 or more operating system to run concurrently
- Distinct from dual boot
  - Only one operating system is active at anytime

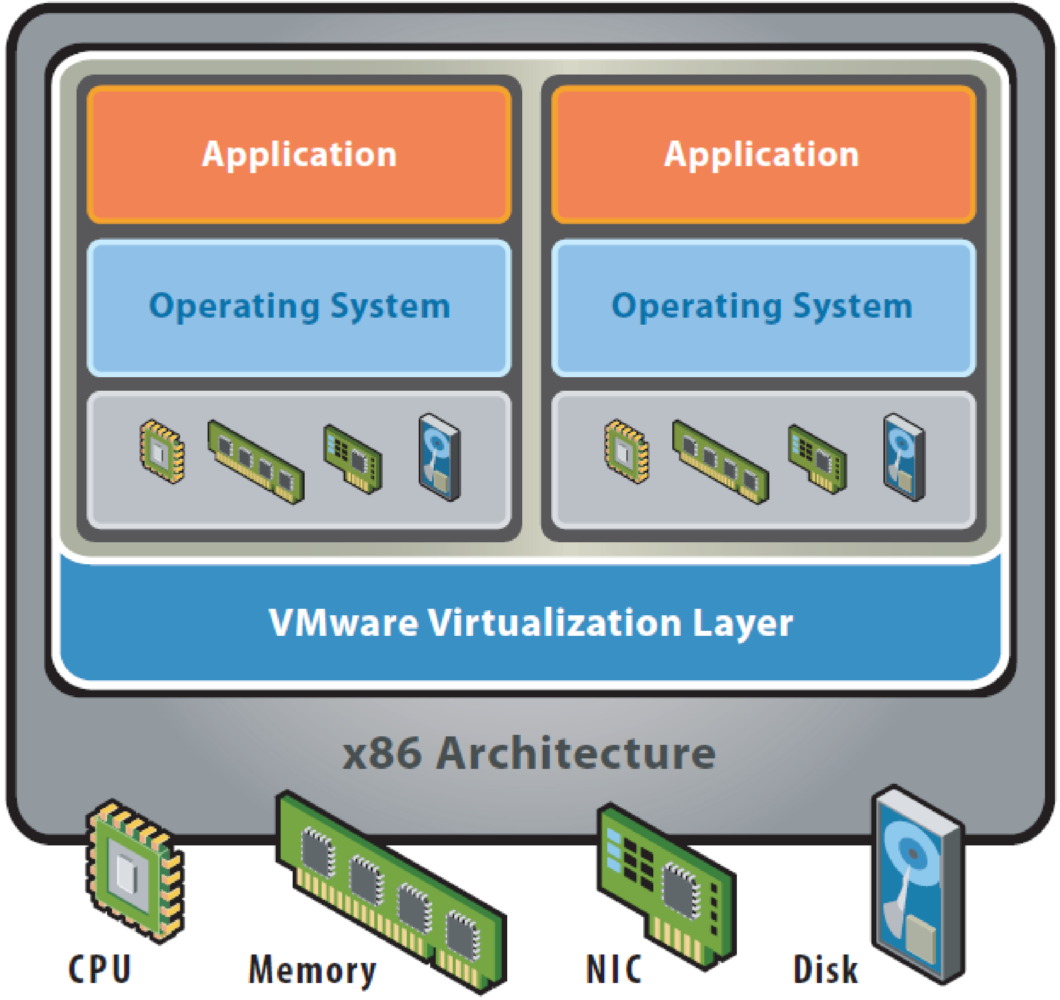
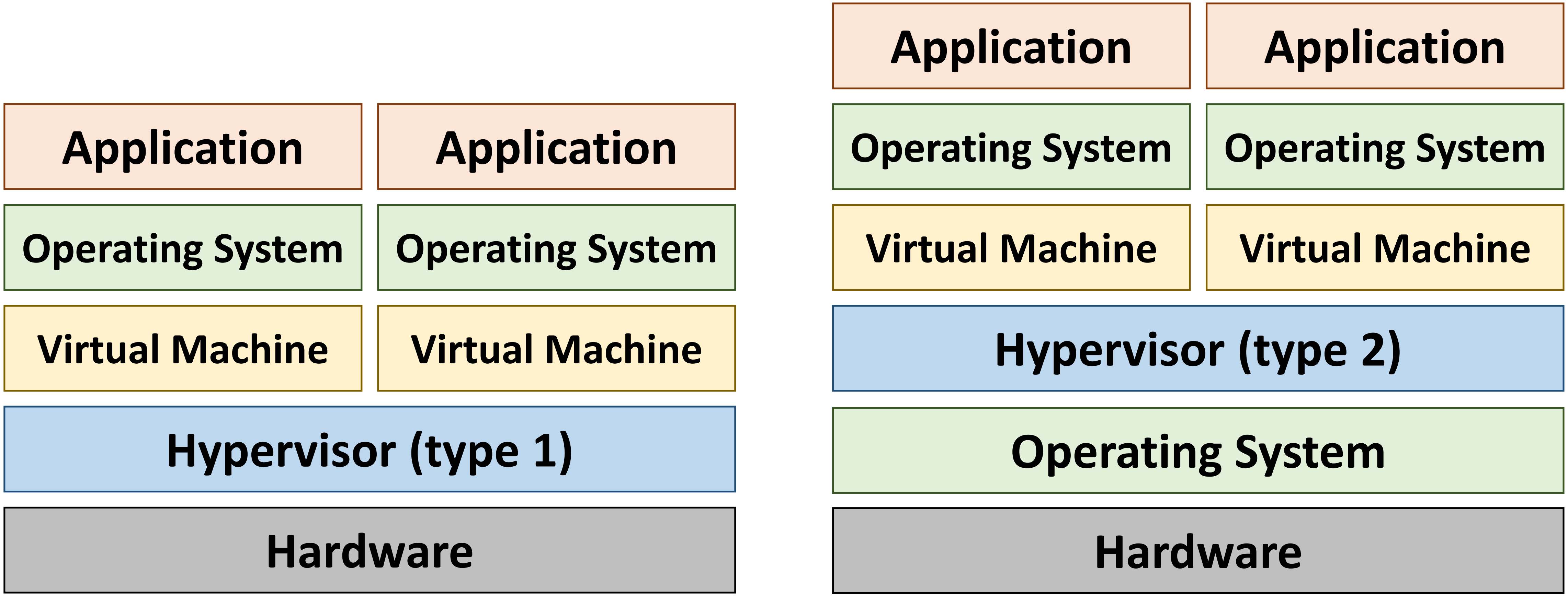


Image from https://www.vmware.com/pdf/virtualization.pdf

Types of Virtualization



Type 1

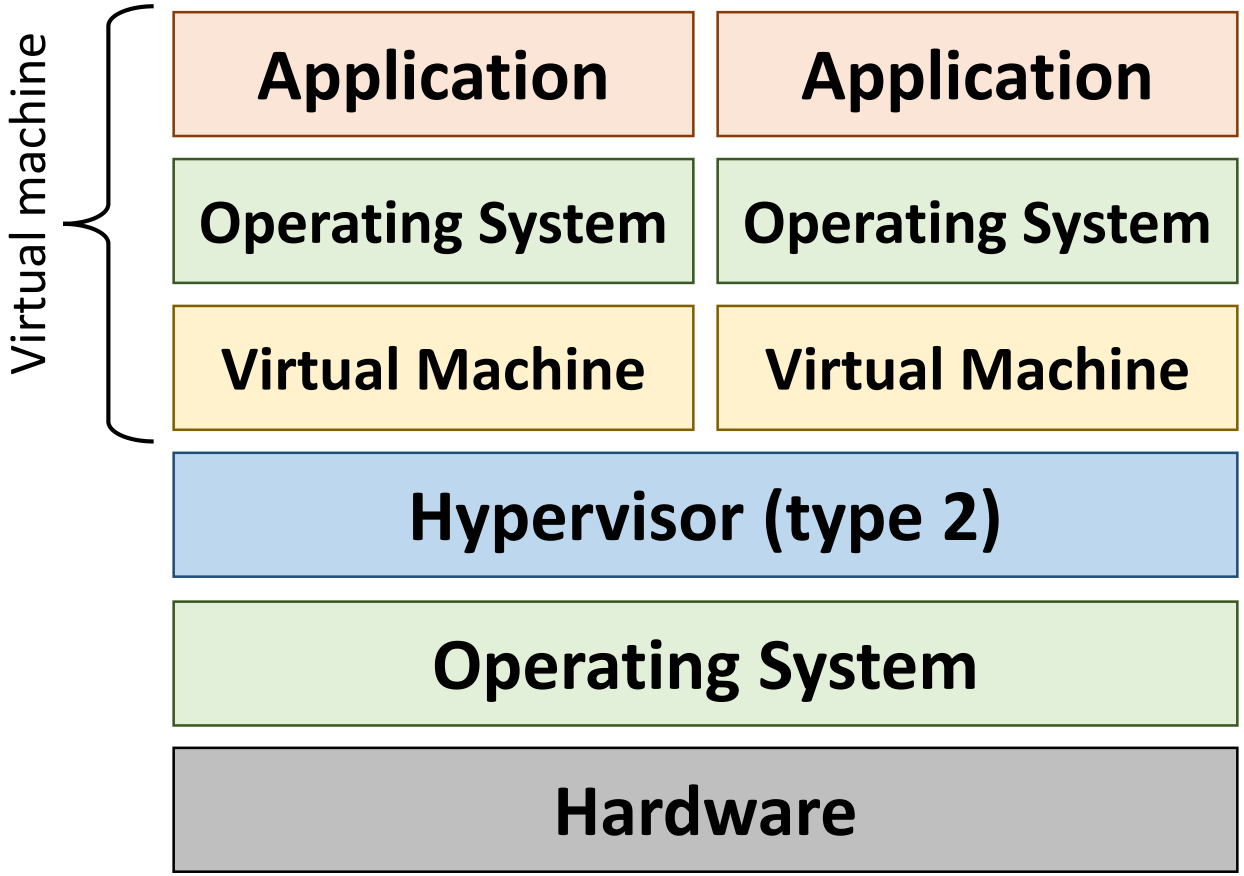
Type 2

Containers

- Another form of virtualization
- Containers virtualizes Linux
  - Uses LXC (Linux Containers)
- Allows multiple Linux applications to run on a single Linux operating system
  - Reduces or eliminates library conflicts

Virtual Machines

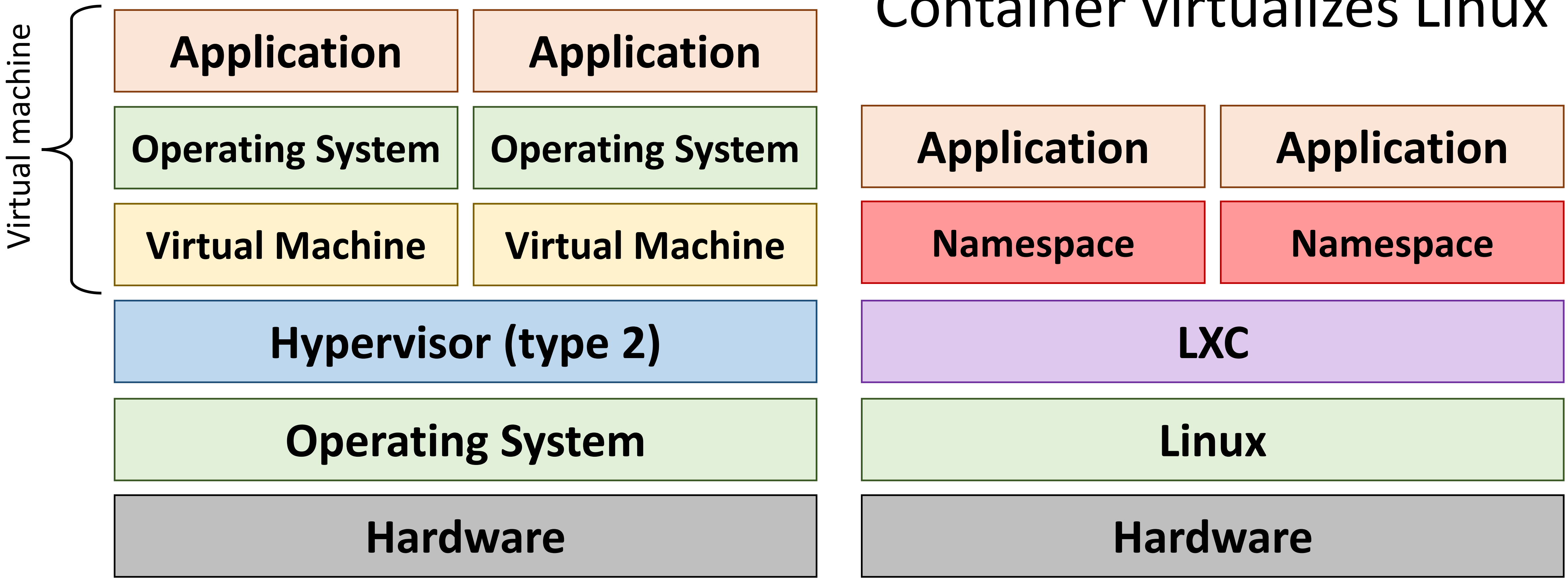
VM virtualizes the hardware



Virtual Machines

VM virtualizes the hardware

Container virtualizes Linux

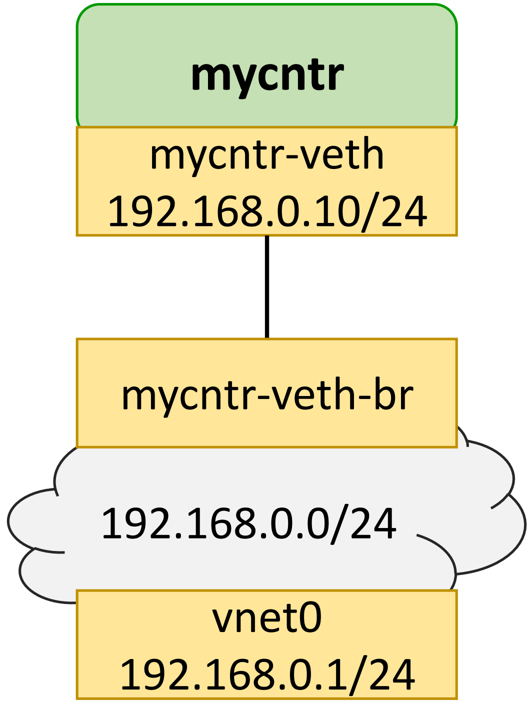


LXC Enabling Technologies

- Namespace
  - Provides isolation
  - Virtualizes Linux resources eg. processes, filesystem, IPC, etc.
- CGroups
  - Allocate resources to the containers
  - Can configure soft and hard limits
- Overlay filesystem
  - Provides a single view of multiple directory by stacking them
  - Provides “copy-on-write” capabilities

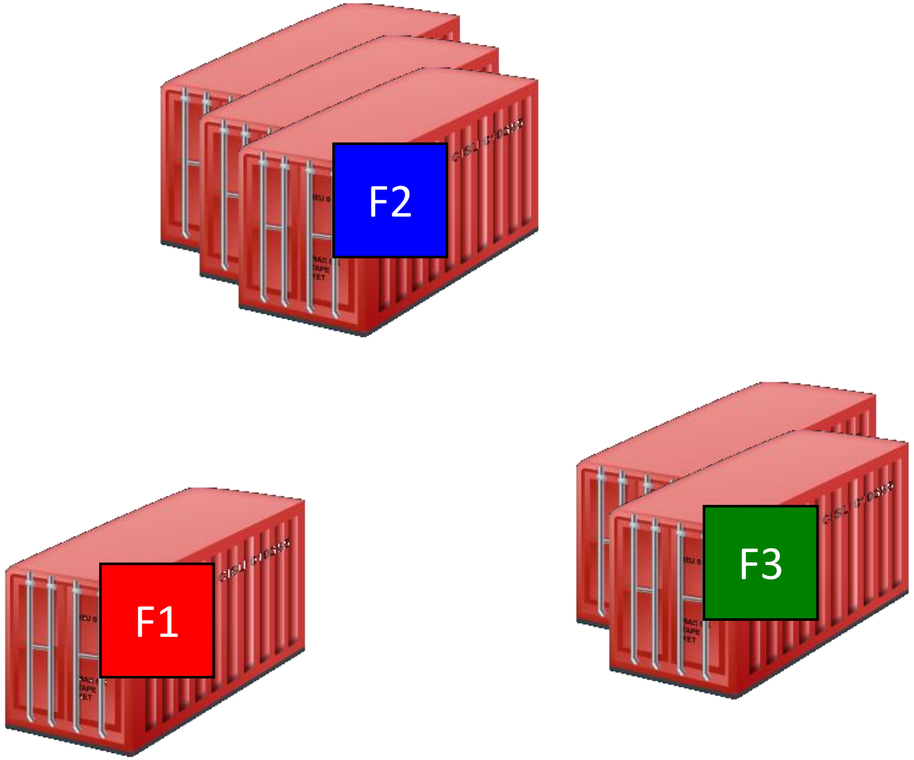
Example - Manually Creating Containers

```
ip link add vnet0 type bridge
ip addr add 192.168.0.1/24 dev vnet0
ip link set dev vnet0 up

ip netns add mycntr
ip link add mycntr-veth dev veth peer name mycntr-veth-br

ip link set dev mycntr-veth netns mycntr
ip -n mycntr addr add 192.168.0.10/24 dev mycntr-veth
ip -n mycntr link set dev mycntr-veth up

ip link set dev mycntr-veth-br master vnet0
ip link set dev mycntr-veth-br up

ip -n mycntr route add default via 192.168.0.1

ip netns exec mycntr node main.js
```



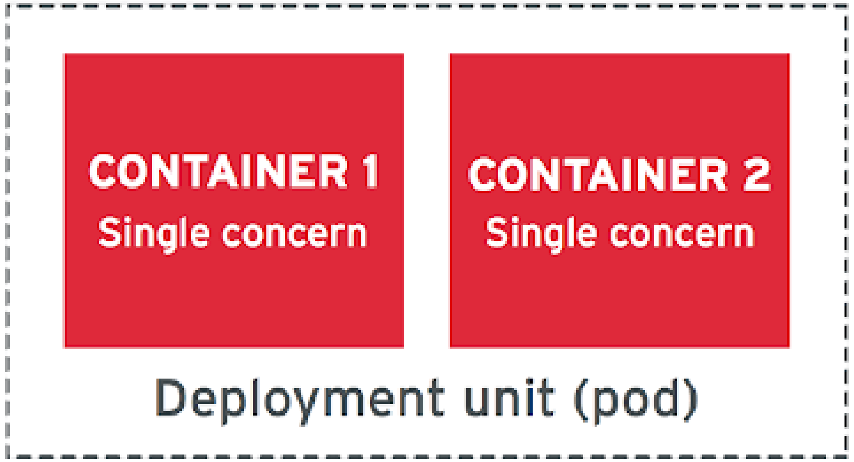
Containers and Micro Services

- Containers are extremely lightweight
- Run each service in its own container
- Scale a service by provisioning more of that service
- Communicate with each other via HTTP or queues

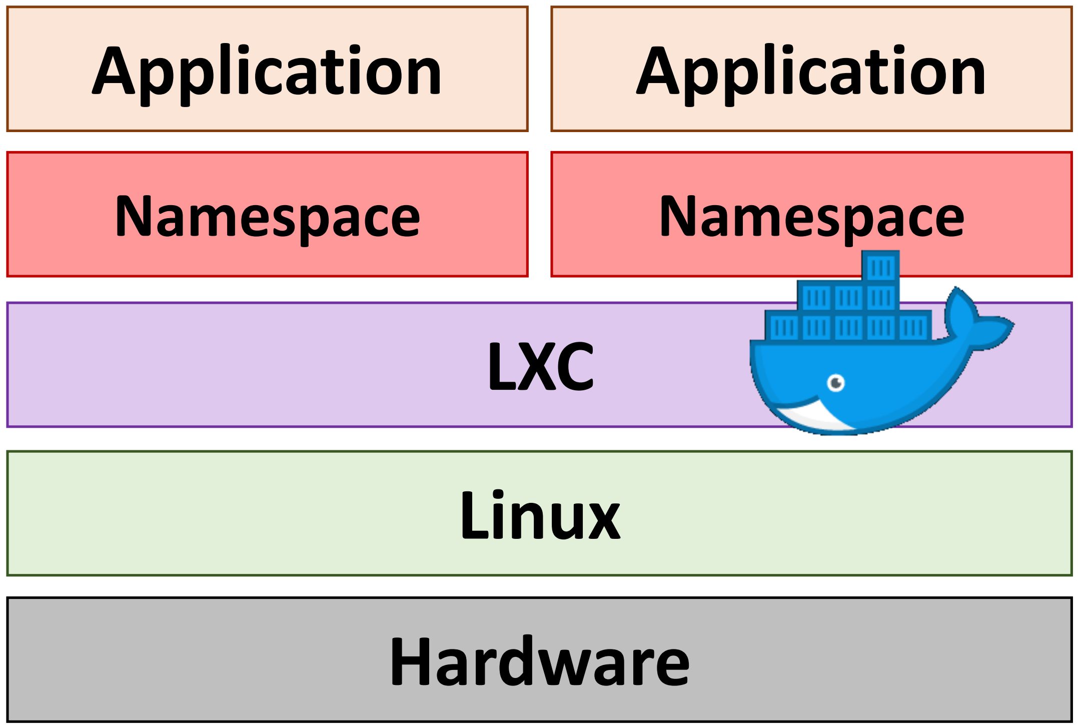


Single Concern Principle

- Containers only deliver one service
  - When a micro service is scope to the appropriate granularity
- Treat containers as service primitives
  - Containers interact with each other to deliver higher level service
- Allow for a container be to swapped out in favour of a better implementation of that service
  - Without disrupting the overall service



Docker



- A set of LXC management tools for creating “containerized” applications
  - Application isolation by namespaces
  - Specific view of the file system
  - Constrained to a set of resources
- A “image” format

Docker Architecture

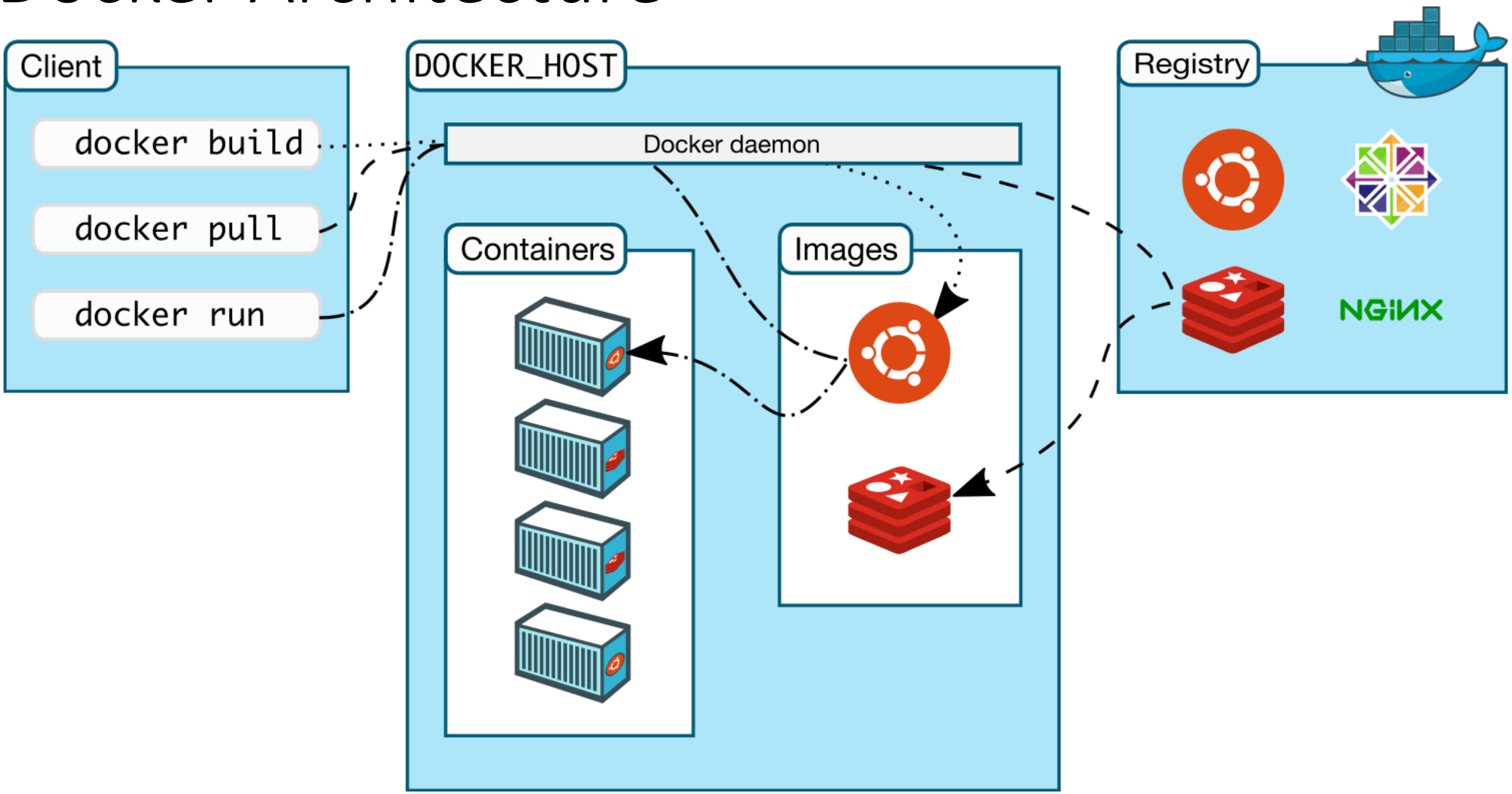
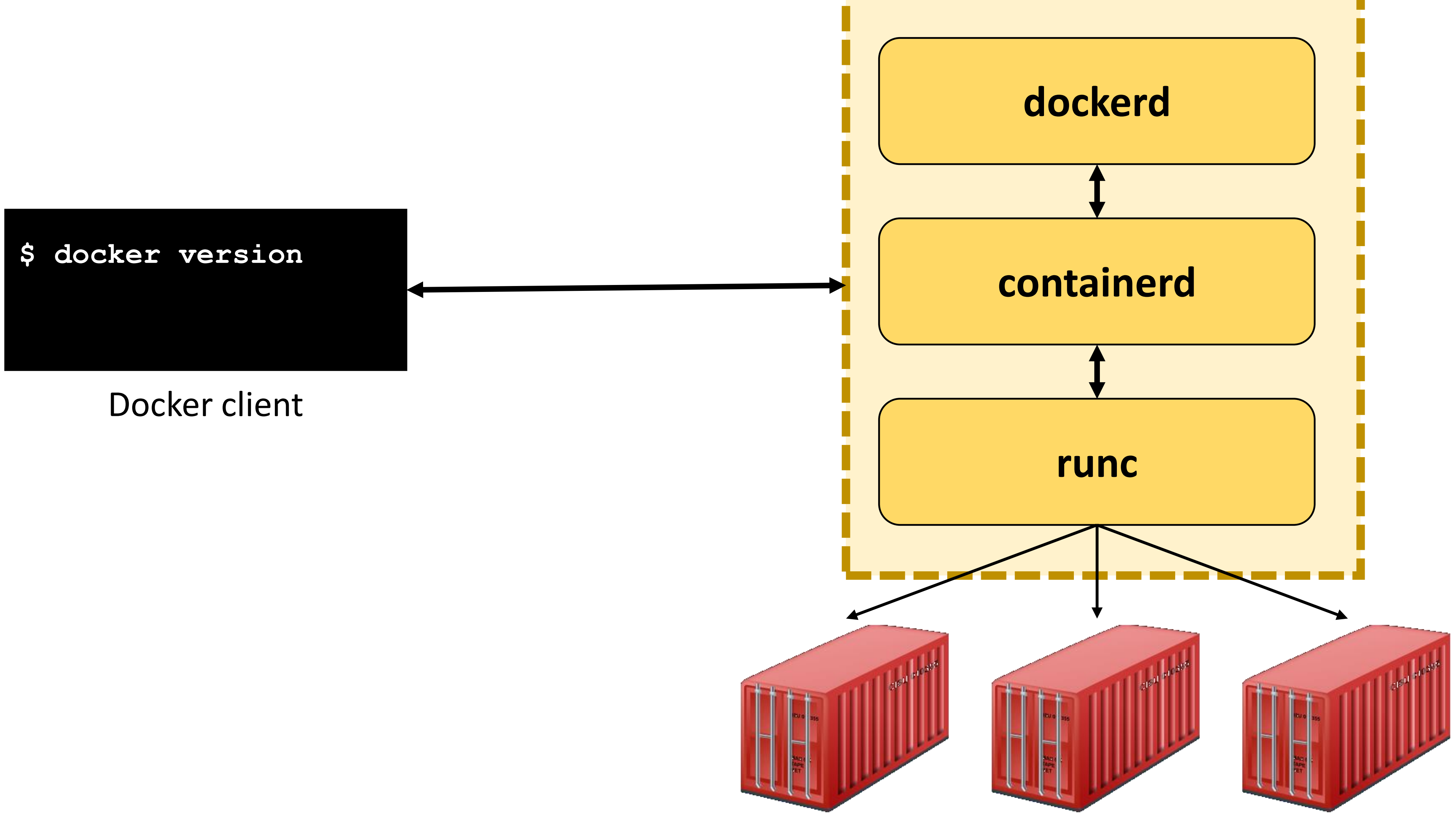


Image from https://docs.docker.com/engine/images/architecture.svg

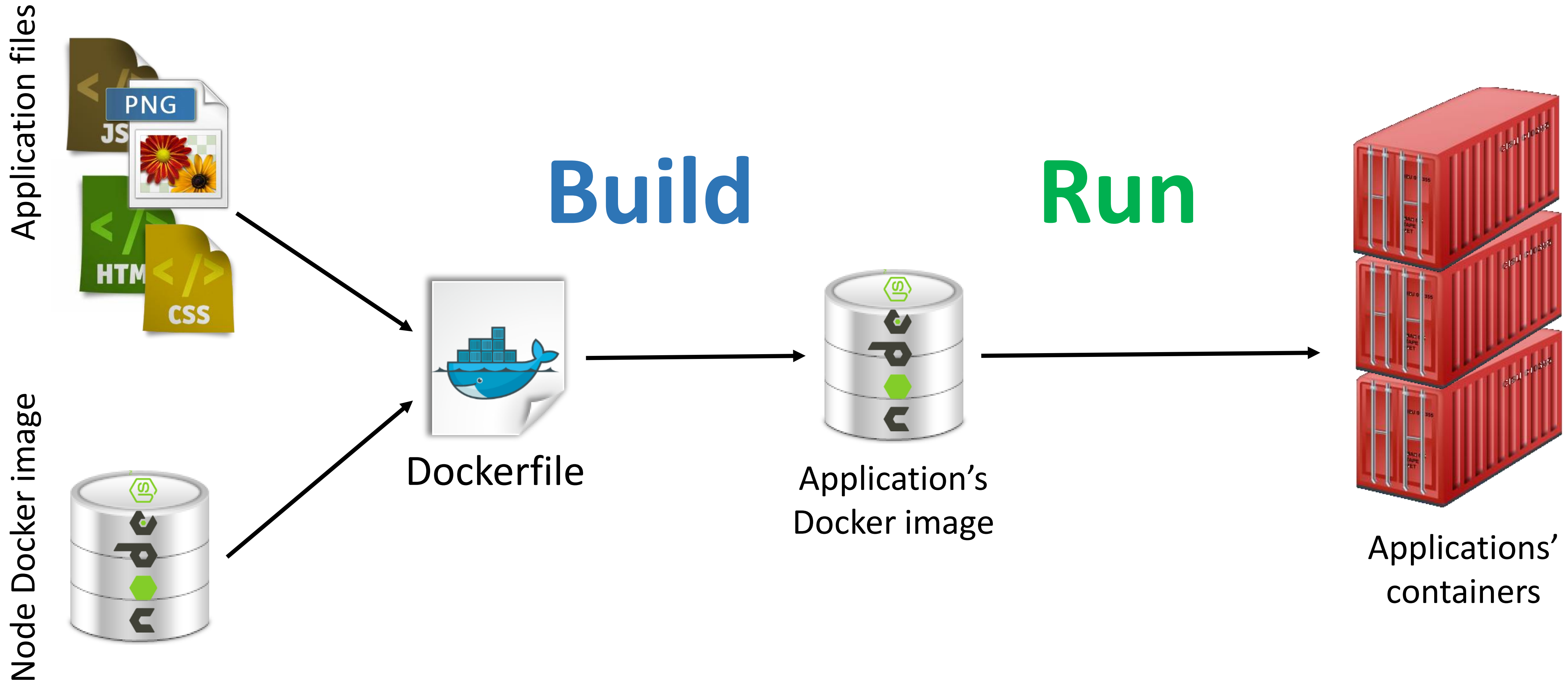
Docker Architecture



Using Docker

- Tool for packaging, deploying and running applications
- An application and all its dependencies are packaged inside a Docker image
  - Ready to run, like a statically compiled binary
- Application/Docker images are consistent and immutable
  - Will run on everywhere that supports Docker

Docker Workflow

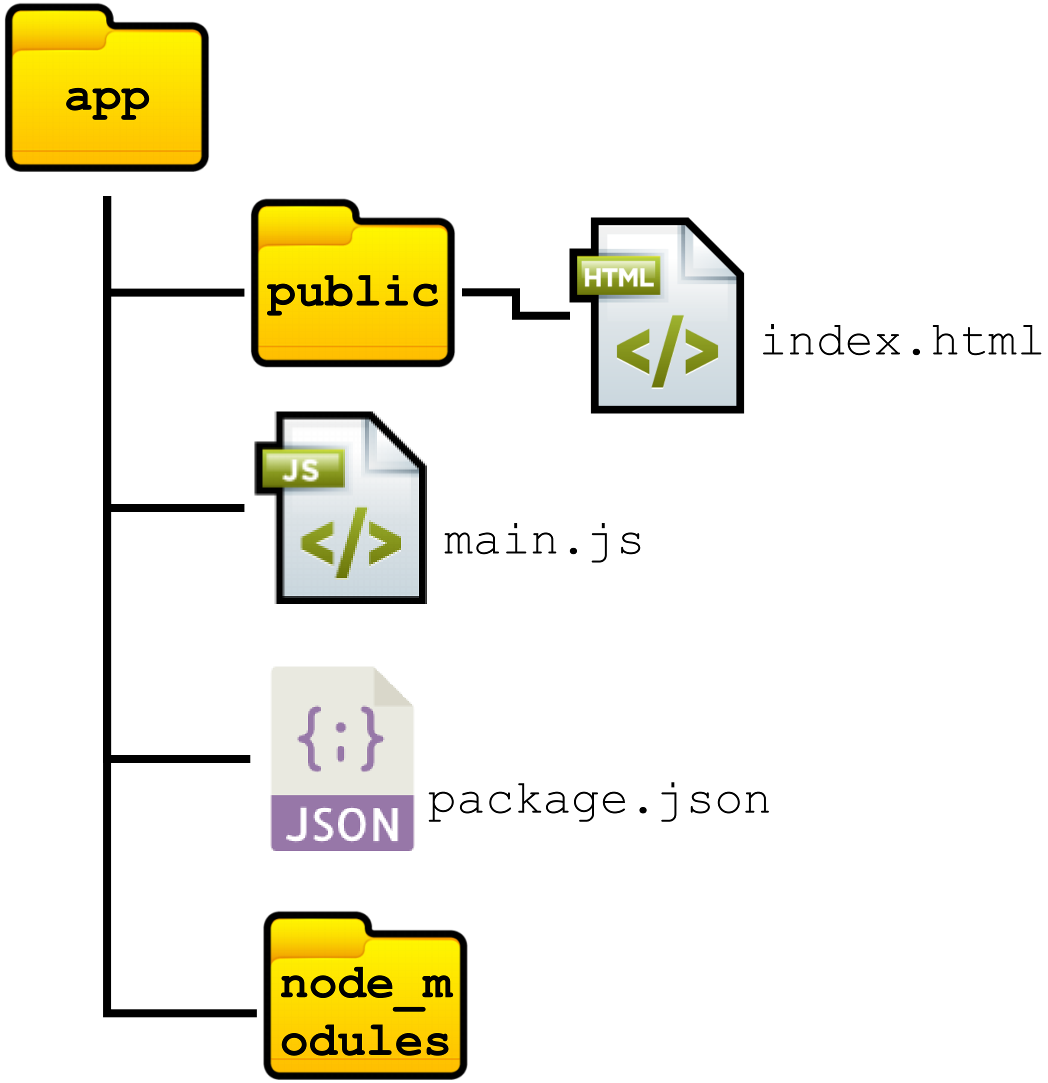


Containerizing an Application

- `Dockerfile` describes how to package an application as a Docker image
  - Like a build file eg. `Makefile`, `pom.xml`
- Describes
  - Application runtime to use
  - Additional packages to install
  - Building the application
  - Executing the application
  - Resources that are needed

Building and Running a Node Application

```
mkdir app
cd app
mkdir public

npm init

npm install --save express

//edit file main.js
//edit index.html in public

node main.js 3000
```



Dockerfile

```
FROM node:16

LABEL "name"="myapp"

ARG APP_DIR=/app

RUN mkdir $APP_DIR

WORKDIR $APP_DIR

ADD main.js .
ADD package.json .
ADD public public

RUN npm ci

ENV APP_PORT=3000
EXPOSE $APP_PORT

ENTRYPOINT [ "node", "main.js" ]
CMD [ "$APP_PORT" ]
```

Use the node image as the base to build the application

Add labels to the image

Build arguments

Run the command to create a directory

Sets the working directory. Like 'cd' into the directory

Add all these files and directories into `$APP_DIR`
Why not `node_modules`?

Installs dependencies

Tell Docker that the application is listening on `$APP_PORT`

Sets the environment variable

Command to execute when container starts

Provide a default for `ENTRYPOINT`

Dockerfile

```
FROM node:16

LABEL "name"="myapp"

ARG APP_DIR=/app

RUN mkdir $APP_DIR

WORKDIR $APP_DIR

ADD main.js .
ADD package.json .
ADD public public

RUN npm ci
```

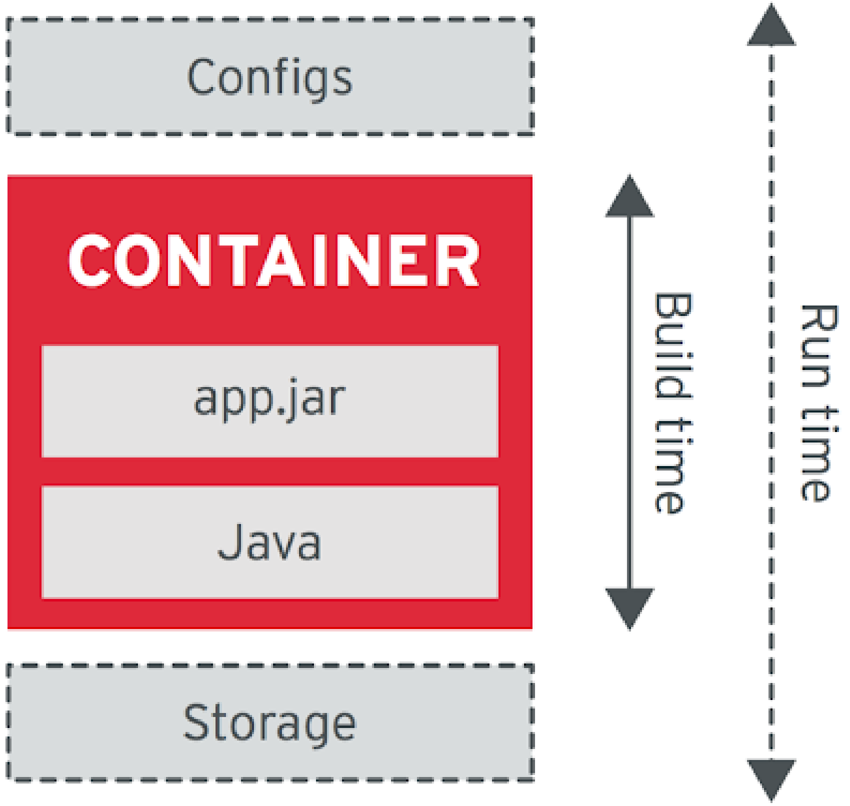
For building the image

```
ENV APP_PORT=3000
EXPOSE $APP_PORT

ENTRYPOINT [ "node", "main.js" ]
CMD [ "$APP_PORT" ]
```

For running the image

Self Containment Principle

- A container should have all dependencies it needs to run the application
  - No other external dependencies
  - Except running on Linux
- Parametrize the things that vary from deployment to deployment
  - Eg. configurations, storage



Building an Image

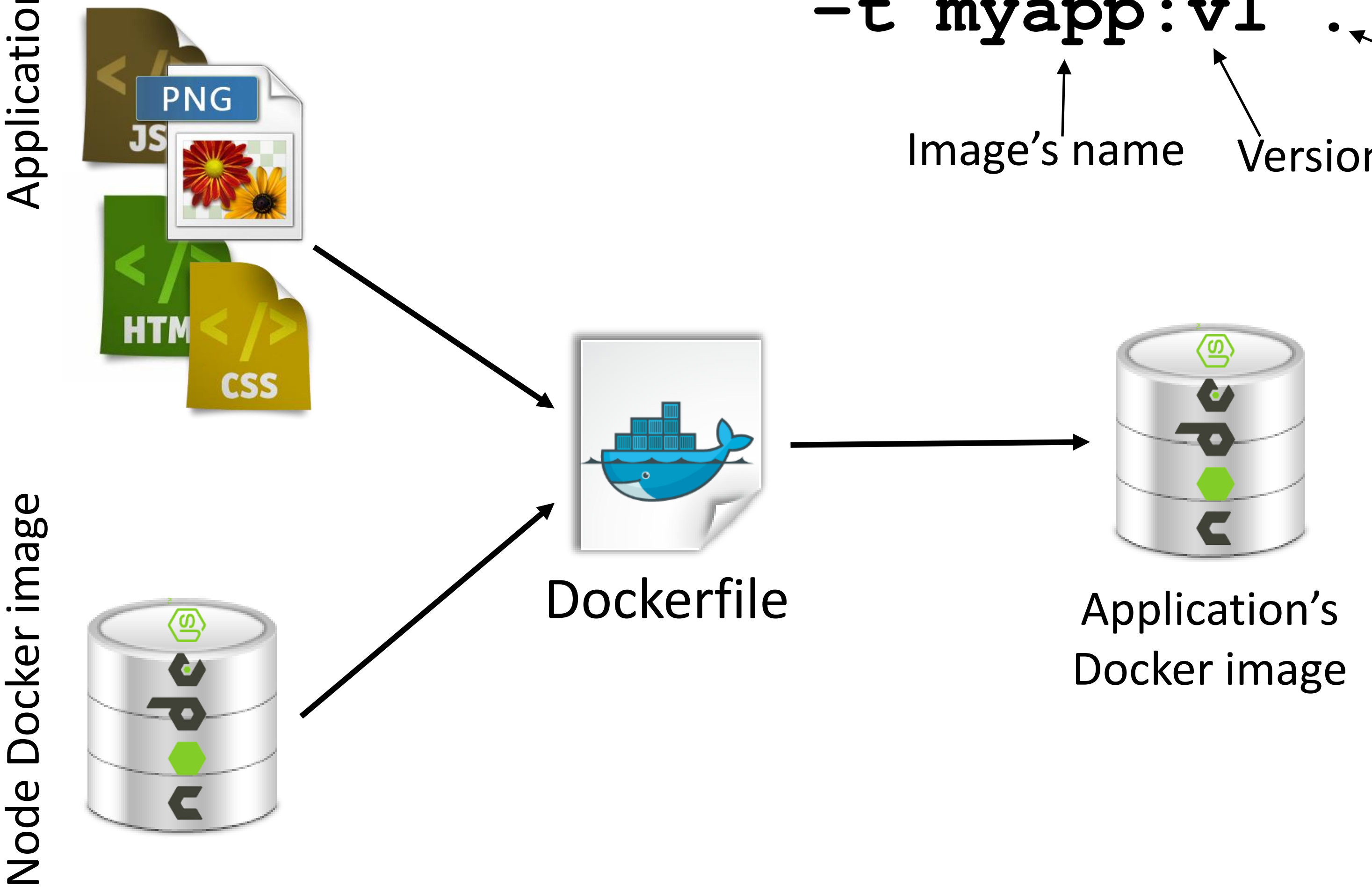
Override the APP_DIR during build

```
docker build --build-arg APP_DIR=/tmp \
  -t myapp:v1 .
```

Image's name

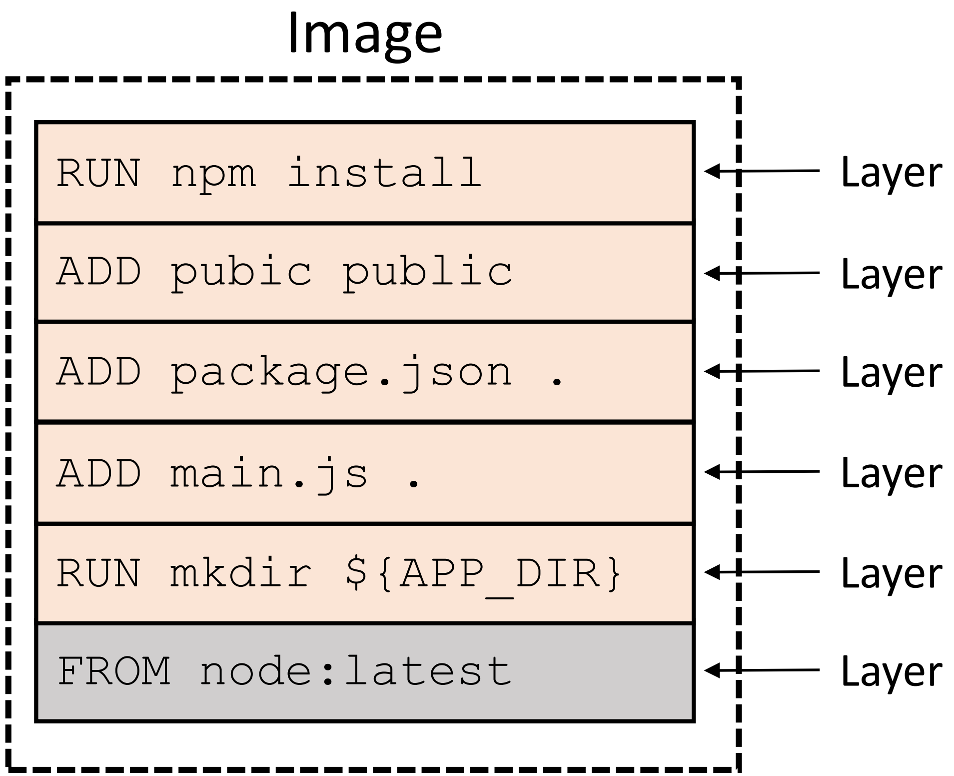
Version

Build context, the location of the files



Docker Image

`docker history myapp:v1`



Running an Image

Run as daemon

Container name

Image name

```
docker run -d -p 8080:3000 --name app myapp:v1
```

Port binding

Docker Host: 192.168.0.10

Network traffic to 192.168.0.10:8080 will be routed to port 3000 in the container

Port Binding

- Container ports are not accessible to the outside world
  - Web applications will not be accessible
  - Will only be accessible to other containers in that network
- Need to specify a port from the host to the container's port
  - Any traffic to the host port will be forwarded to the mapped container port
- Port binding defines this relationship
  - When creating a container

```
docker run -d -p 8080:3000 --name app myapp:v1
```

Port binding

Environment Variables

Set environment variables. Use additional `-e` to set multiple variables

```
docker run -d -p 8080:5000 -e APP_PORT=5000 \
     --name app myapp:v1
```

Container and Image Management

- List all running containers

```
docker ps
```

- Stop a container

```
docker stop mycontainer
```

- Start a container

```
docker start mycontainer
```

- Delete a container

```
docker rm mycontainer
```

- List all images

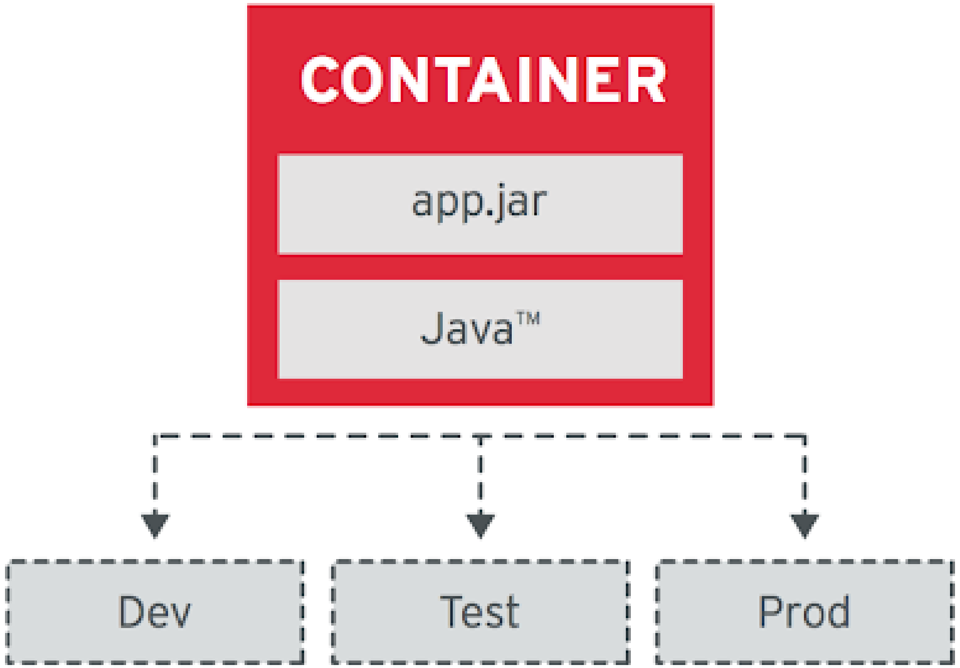
```
docker image ls
```

- Delete an image

```
docker rmi myimage
```

Image Immutability Principle

- Images are immutable
- Same image should be use for dev, test and production
  - Only configuration should change
- Do not create snowflakes
  - Exec into a container and patch it
- Should rebuild the image and redeploy



High Observability Principle

- Container are black boxes
- Need to define a standard interface for the container runtime to observe its health
- Suggested observables
    - Readiness - when an application to serve; may be different from when the container is ready
        - Called once at startup
    - Liveness - is the application still alive
        - Called multiple times over the lifetime of the container
- Tracing - allow a request to be traced - OpenTracing
    - Eg. the amount of time spend in a particular database query
- Logs

Lifecycle Conformance Principle

- Receive events from the runtime
  - Inform the container of what is happening
- Application within the container should handle those events

Example of High Observability - Application

```
const pool = mysql.createPool({ ... })
const app = express();
let ready = false;

app.get('/ready', (req, resp) => {
  resp.status(ready? 200: 400).end();
})

pool.getConnection((err, conn) => {
  conn.ping((err) => {
    ready = !err;
  })
})
process.on('SIGTERM', () => {
  //Received SIGTERM - perform clean up
})
```

Readiness probe. Returns 200 - 399 if the app is ready.
Can double as liveness probe

Clean up before the container is removed

Example High Observability - Docker

```
FROM node@sha256:af23..

...

HEALTHCHECK --interval=30s --timeout=5s --retries=3 \
   CMD curl -s -f http://localhost:${APP_PORT}/ready > /dev/null || exit 1

ENTRYPOINT [ "node", "main.js" ]
CMD [ "$APP_PORT" ]
```

Time between health check probe

Number of failed attempts for a container to be considered unhealthy

Returns 0 if successful. Pass health check

Failed health check

Persistent Data

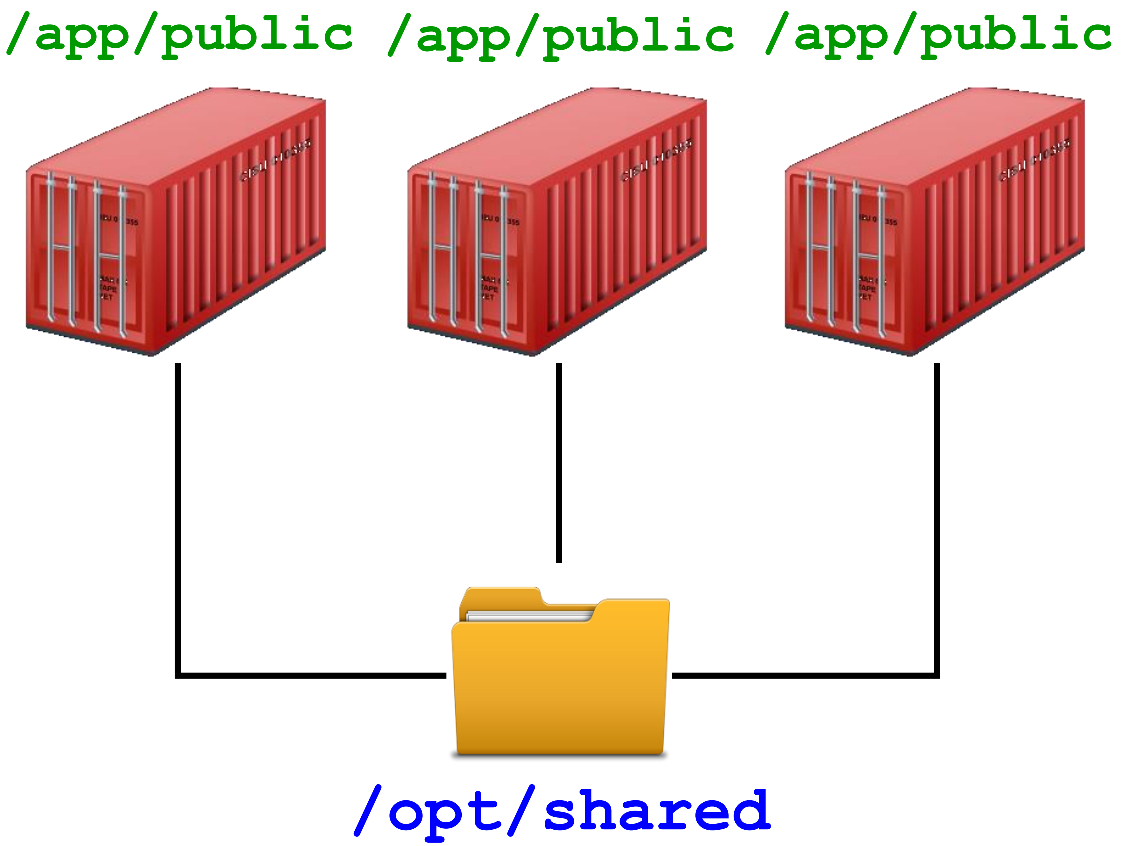
- Containers are ephemeral
  - Nothing in a container is persisted when a container is removed or dies
  - Eg. Access logs captured by Morgan will not be retained
  - Eg. MySQL database
- Persistent data has to be externalized
  - Written to storage volumes outside of the container
  - When the container is deleted, the data is not delete as well
- Two ways of mapping external storage into Docker
  - Mount a directory from the Docker host into the container
  - Define a Docker volume and mount the volume into the container

Mounting a Local Directory

```
ENV APP_PORT=3000 APP_DIR=/app
...

VOLUME ${APP_DIR}/public

EXPOSE ${APP_PORT}
...
```

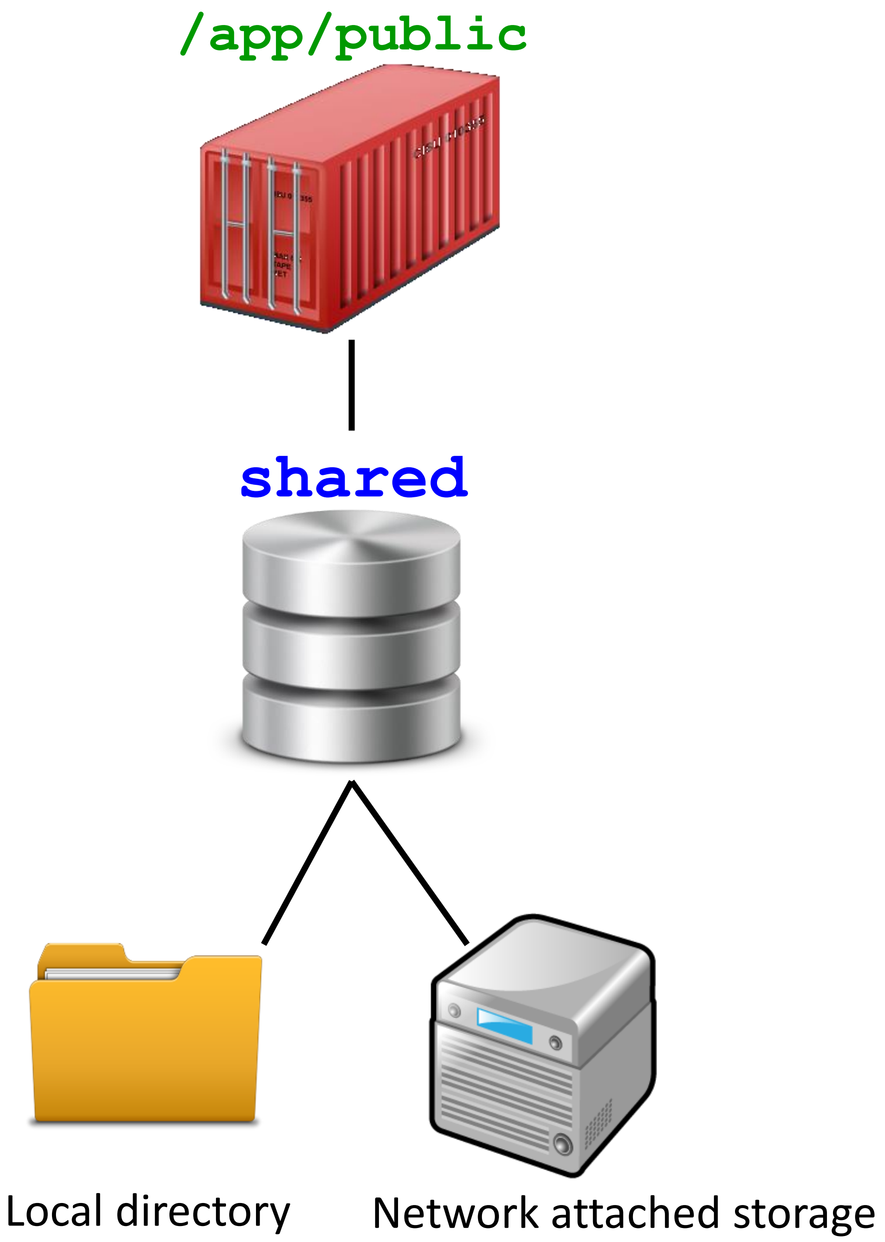
Define a mount point in the container

```
docker run -d -p 8080:3000 \
   -v /opt/shared:/app/public \
   --name app myapp:v1
```



Volumes

- Volumes is an abstraction of storage in Docker
  - Different plugins provides storage features
- Properties of volume
  - Local or remote (network attached)
  - Storage type can be block, file or object
    - Block – AWS EBS
    - File – NFS, SMB
    - Object – AWS S3, GCP Cloud Storage



Volume Management

- Create a volume

```
docker volume create myvol
```

- List available volumes

```
docker volume ls
```

- Display the properties of a volume

```
docker volume inspect myvol
```

- Delete a volume

```
docker volume rm myvol
```

Creating and Mounting a Volume

```
docker volume create shared

docker run -d -p 3000-3100:3000 \
 -v shared:/app/public --name app0 myapp:v1
```

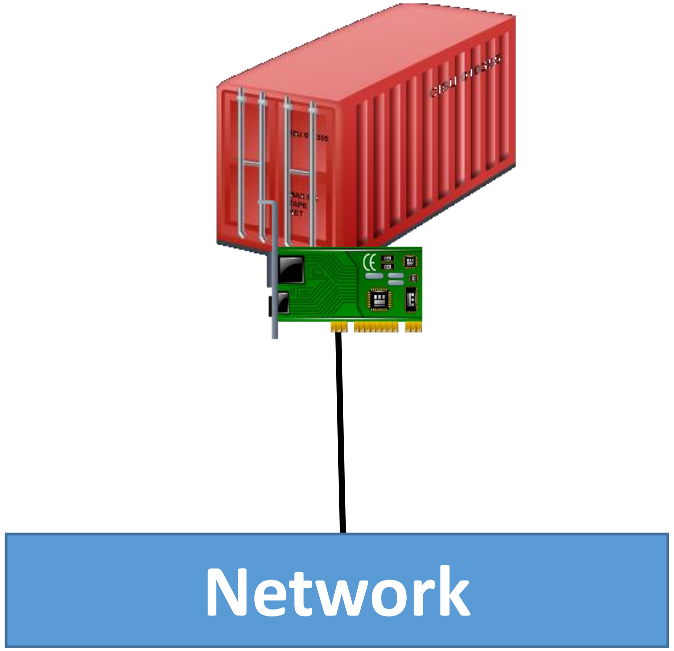
Volume name
without the leading /

Process Disposability Principle

- Containers are ephemeral
  - Can die due to underlying hardware
  - Gets reschedule somewhere else - orchestration
- Externalize your data otherwise its gone
- Design containers to be nimble
  - Quick startup
  - Fail fast

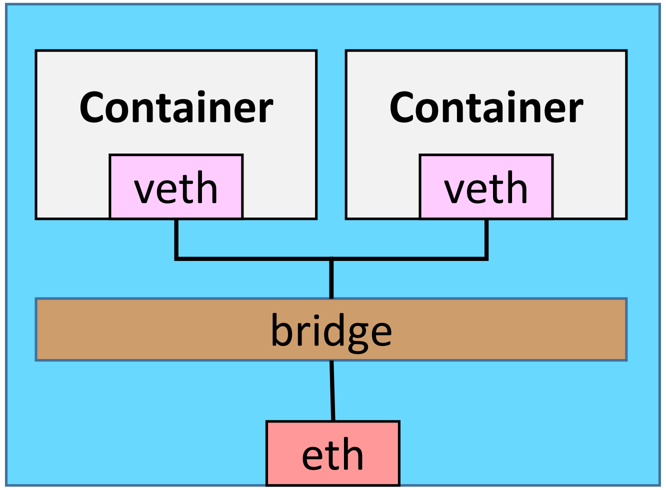
Networking

- Each container get their own
  - Network stack
  - Network interface
    - Virtual network interface (veth)
- Containers connect to their own isolated network
  - Software implementation of 802.1d bridge
- Network are configurable



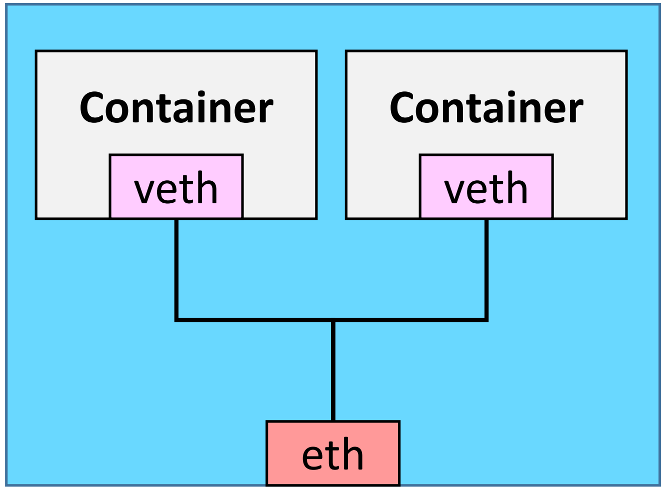
Docker Network - Bridge

- Allows containers to connect to the same bridge network to communicate
  - Docker creates a default bridge network called `bridge` that all containers are plumbed to if you did not specify any network
  - On Windows bridge is called nat



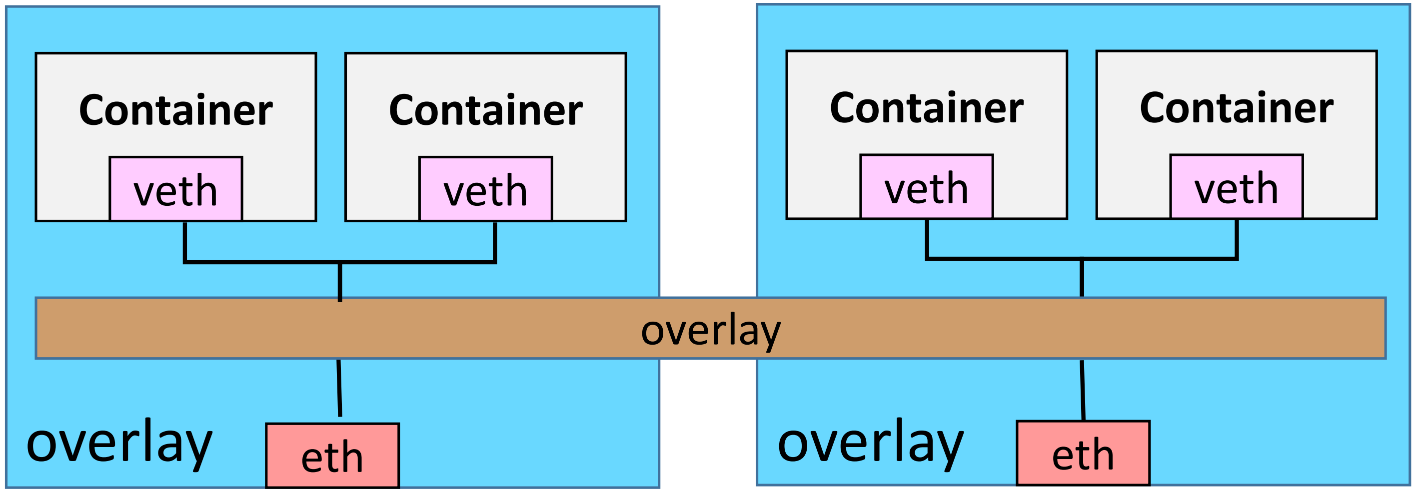
Docker Network - Host

- Container connects into the host's network



Docker Networking - Overlay

- Allows multiple Docker daemon/host to communicate with each other by creating a network on top of (overlay) of the host network



Attaching to Network

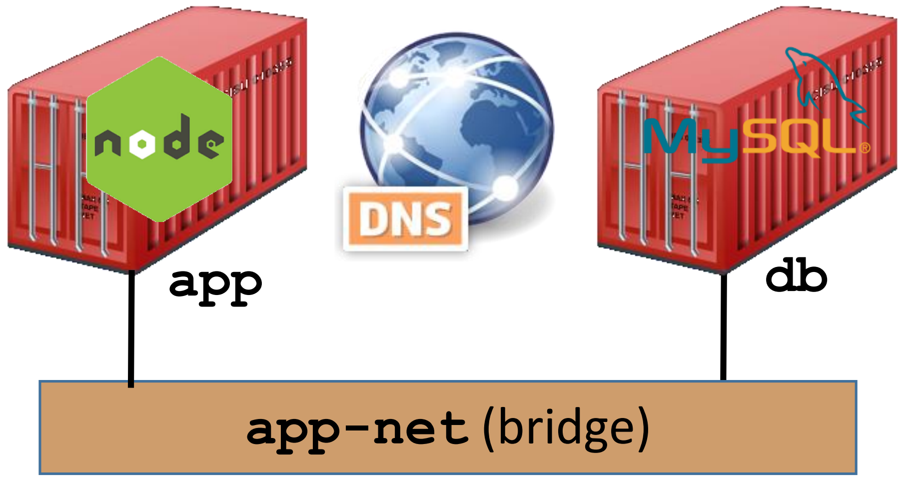
Plumb the container to bridge network

```
docker run -d -p 8080:3000 --name app myapp:v1
```

```
docker network create -d bridge mynet

docker run -d -p 8080:3000 --network mynet \
    --name app myapp:v1

docker network inspect mynet --format '{{json .Containers}}'
```

Service Discovery

- Docker creates an internal DNS service for User created bridge network
    - Containers connected to the network can communicate via their container name, the `--name` parameter
- Default bridge network, bridge, does not support name resolution via Docker's internal DNS
    - Only user defined bridge networks are supported



Network Management

- Create a network

```
docker network create -d bridge mynet
```

- List available volumes

```
docker network ls
```

- Display network properties

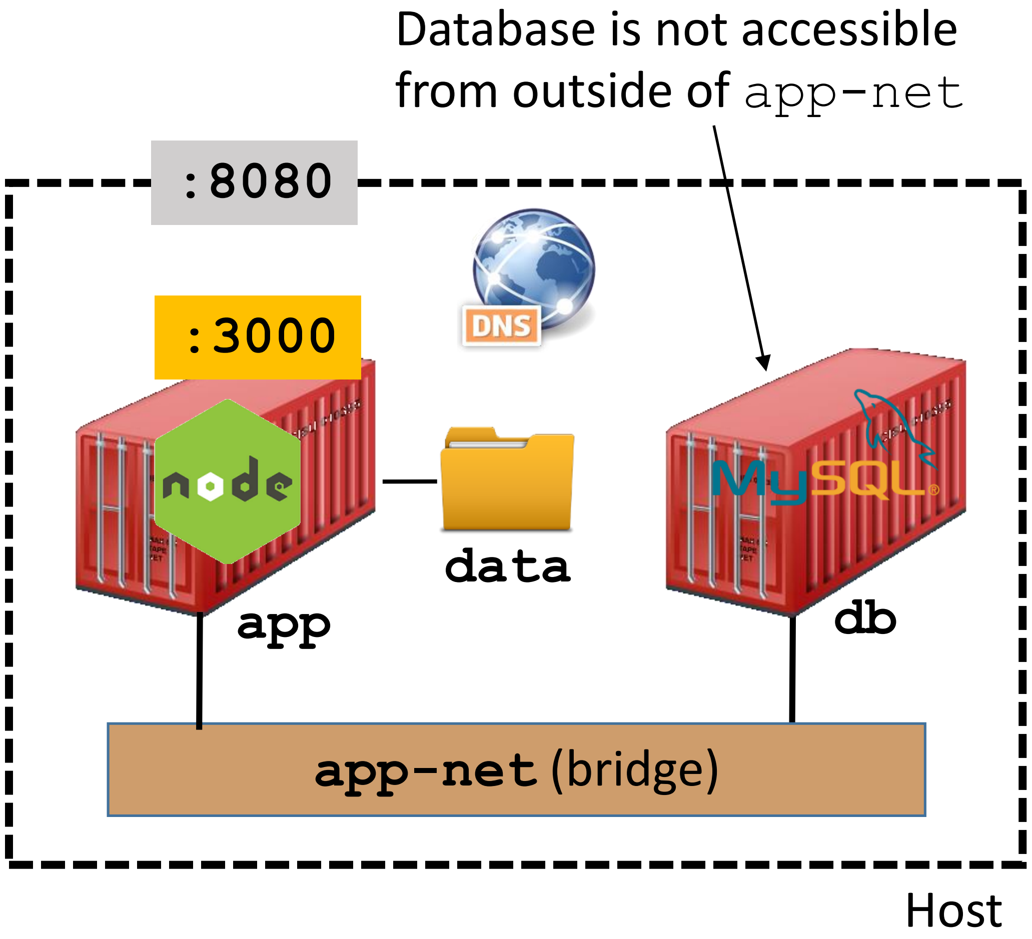
```
docker volume inspect mynet
```

- Delete a volume

```
docker volume rm mynet
```

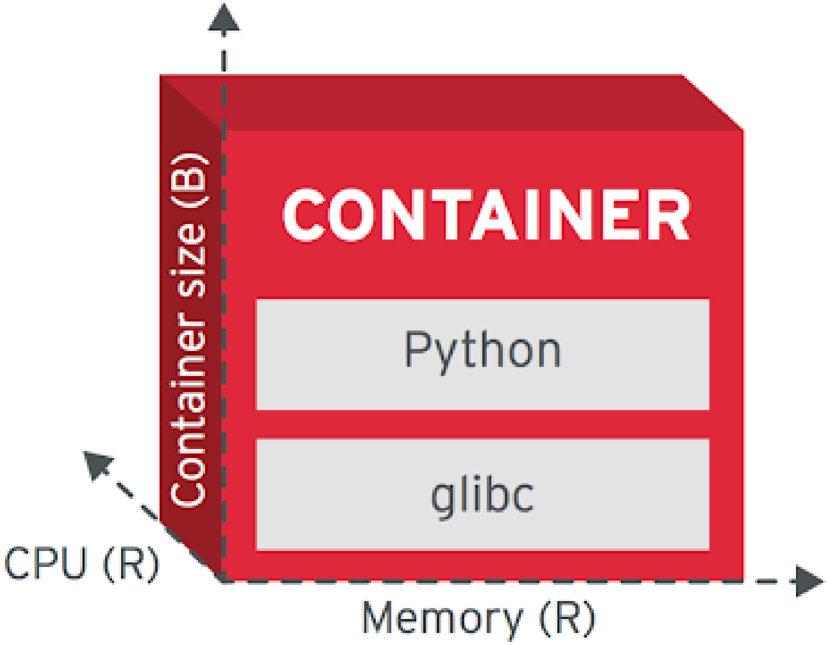
Deploying Application Stack with Docker

```
docker create network \
  -d bridge app-net

docker create volume data

docker run -d \
  --network app-net \
  --name db northwind-db:v1

docker run -d -p 8080:3000 \
  -v data:/app/public \
  --network app-net \
  --name app nortwind-app:v1
```



Runtime Confinement Principle

- Many containers may be running on a single host
- Need to sandbox the containers for resource usage
  - Eg. erroneous application don't hog all the resource




```
docker run -d -p 8080:8080 \
   --cpu-shares=100 \
   --memory=16m \
   --blkio-weight=100 \
   ...
```

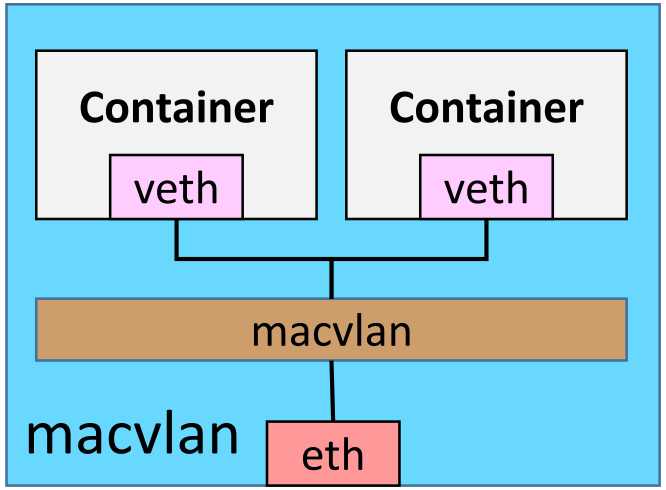
Between 1 - 1024

Between 10 - 1000

Appendix

Docker Networking - Macvlan

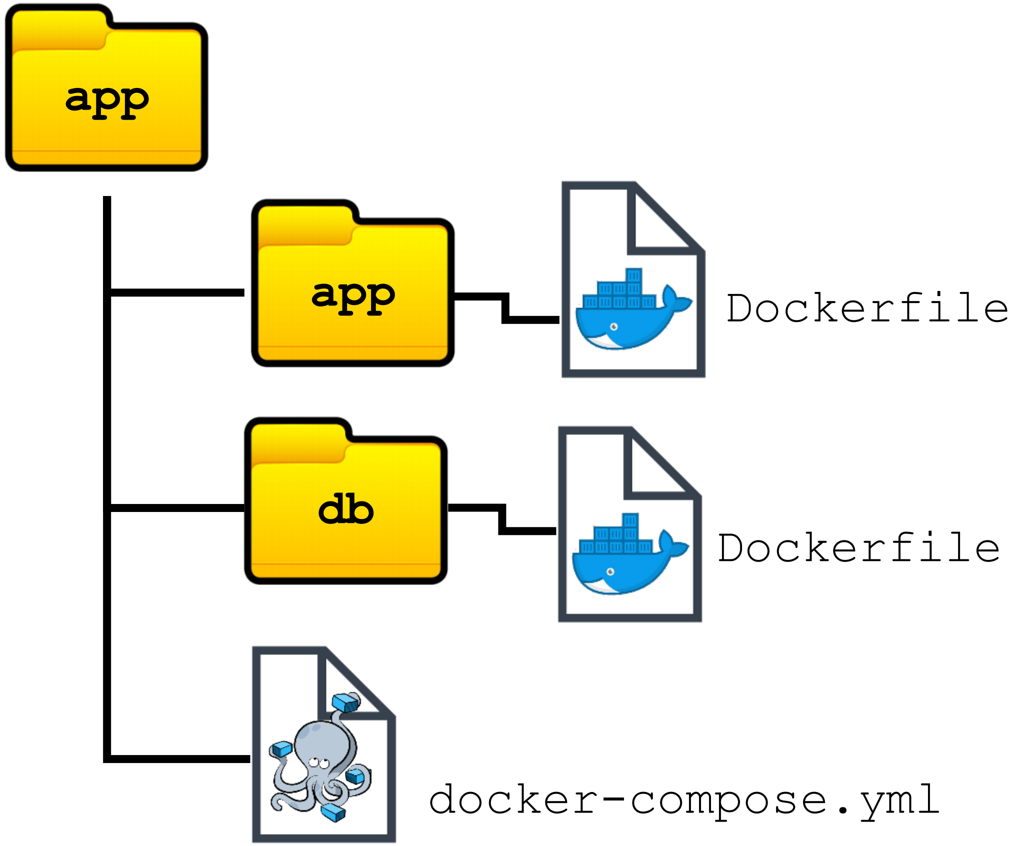
- Allows containers to be directly connected to the physical network
  - Each container will have their own IP address
  - Containers appear as independent systems on the physical network

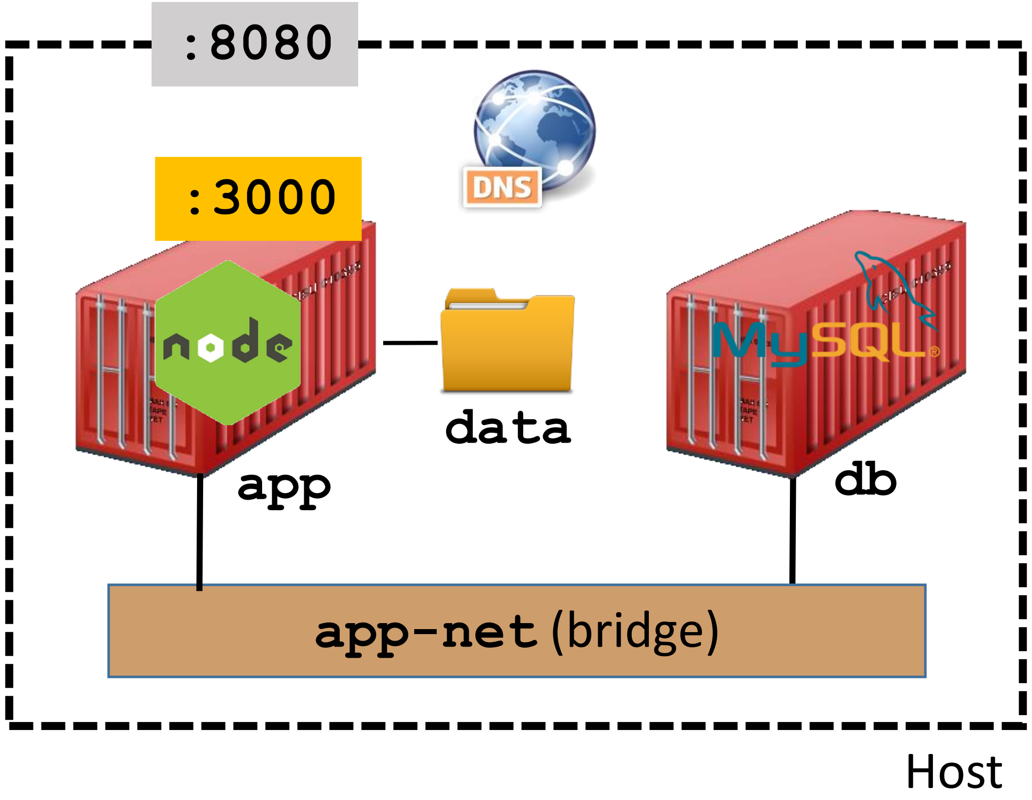


Docker Compose

- Tool for defining and running multi-container application
  - Instead of staring each container individually
- Easily bring up or tear-down entire application stack
- Prioritize resource creation
  - Eg. create networks first before containers
- Docker compose file docker-compose.yml consist of the following 3 main parts
  - services – define one or more containers. Each container is considered a service with a name that can be used by other containers for communication
  - networks – define the network to be created
  - volumes – define volumes

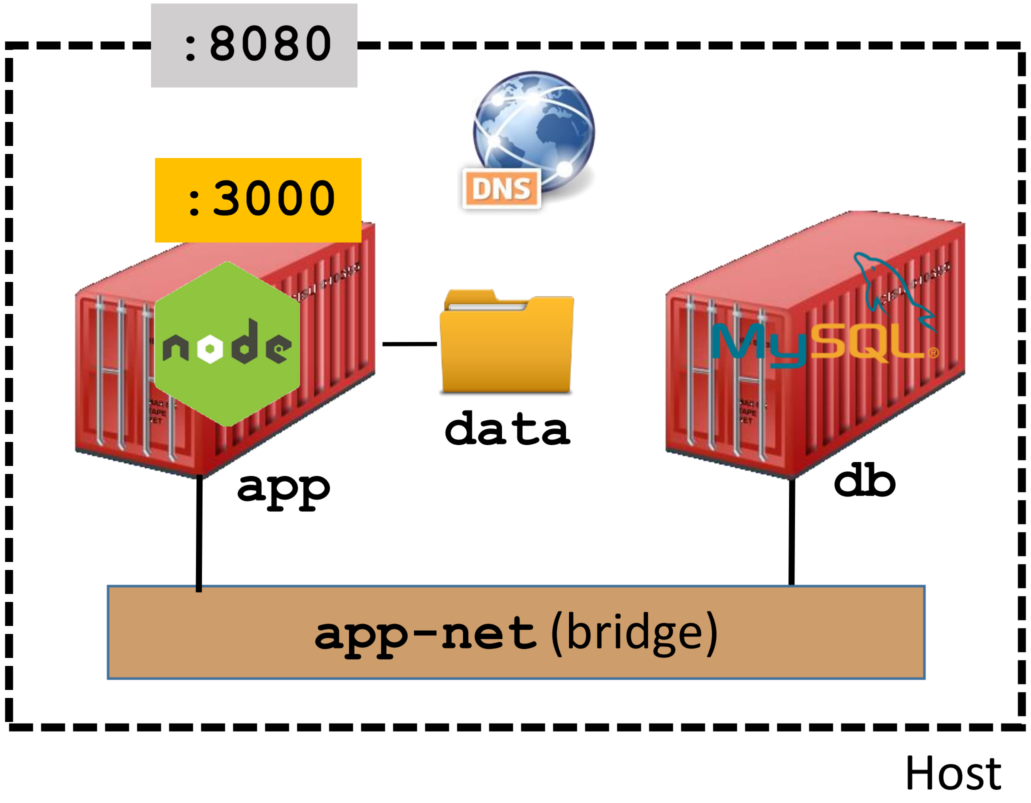
docker-compose.yml





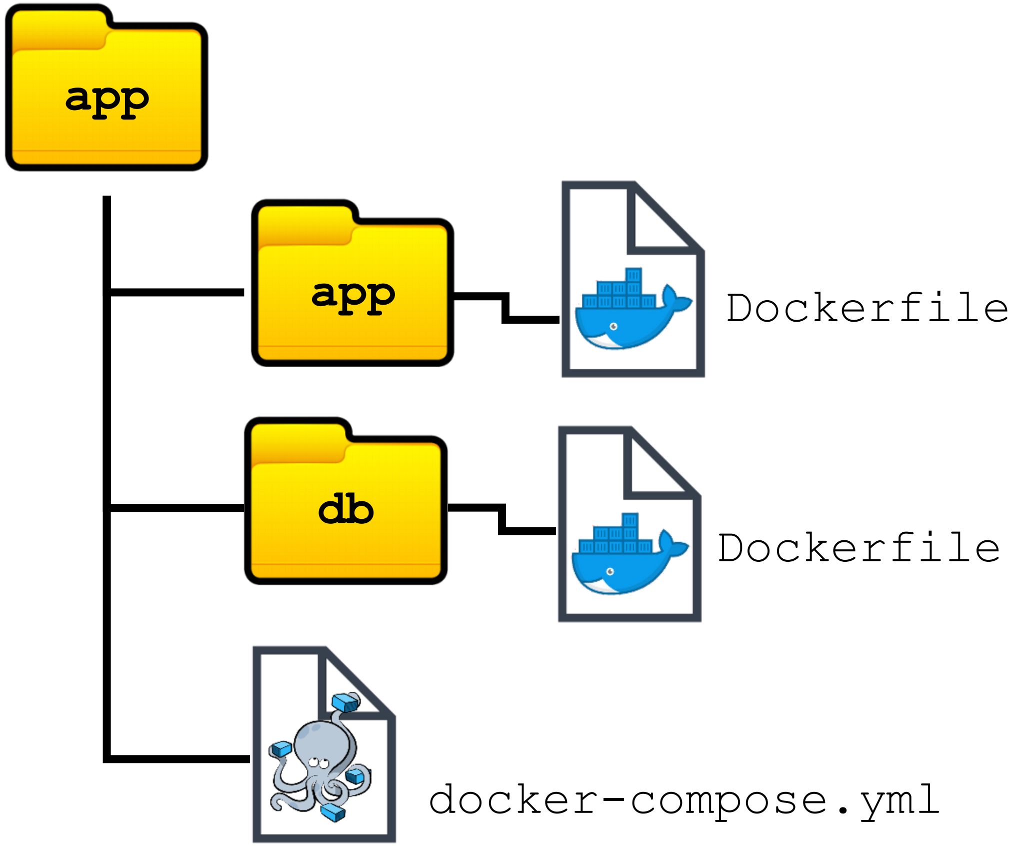
docker-compose.yml

```
version: '3'

volumes:
   data:

networks:
   app-net:
```



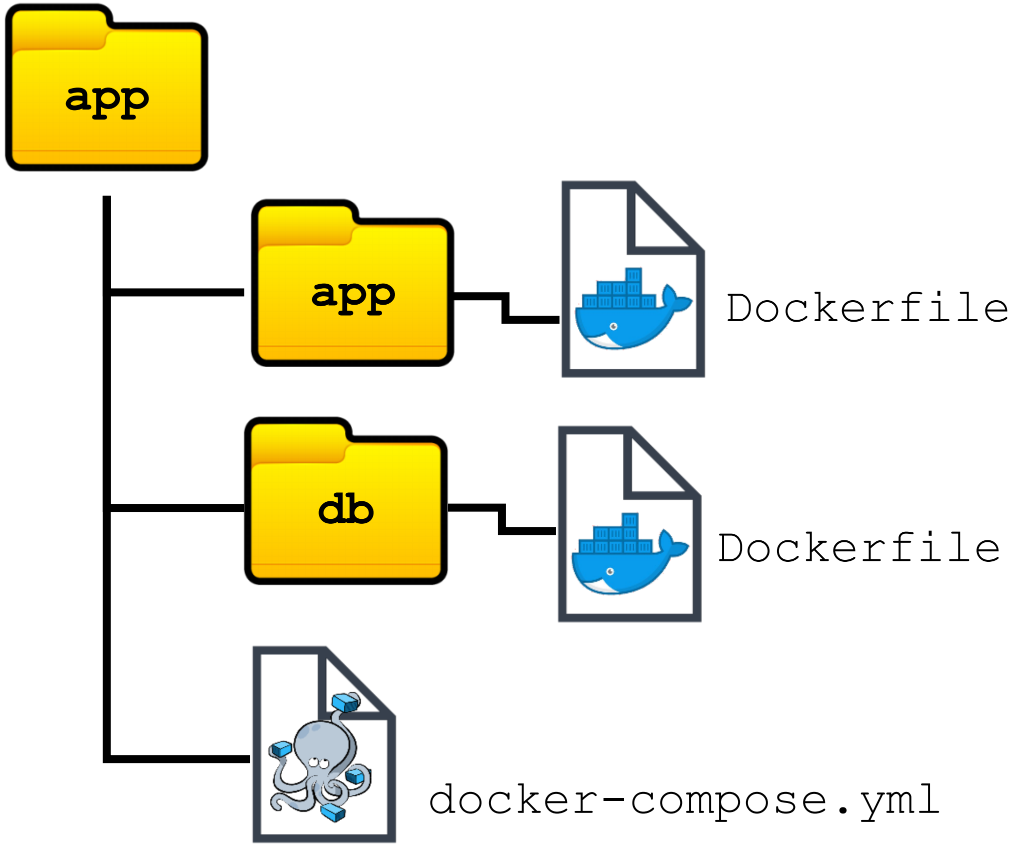
docker-compose.yml

```yaml
...
services:
  app:
    image: northwind-app:v1
    build:
      context: ./app
    environment:
      - APP_PORT=3000
      - DB_HOST=db
      - DB_USER=root
      - DB_PASSWORD=secret
    ports:
      - 8080-8090:3000
    volumes:
      - data:/app/public
    networks:
      - app-net
```



docker-compose.yml

```yaml
services:
  ...
  db:
    image: northwind-db:v1
    build:
      context: ./db
    environment:
      - MYSQL_ROOT_PASSWORD=secret
    networks:
      - app-net
```



Docker Compose

- Starting a Docker application stack

```
docker-compose up -d
```

- Tearing down a Docker application stack
  - Will remove all containers and network
  - Will not remove volumes and images

```
docker-compose down
```

- Stop the application

```
docker-compose stop
```

- Start the application

```
docker-compose start
```

- Build the images in the stack

```
docker-compose build
```